Дискета на 3-ей полосе обложки

ЭД БАРФИЛД
БРАЙЕН УОЛТЕРС

ПРОГРАММИРОВАНИЕ "КЛИЕНТ-СЕРВЕР" В ЛОКАЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ СЕТЯХ

LOCAL AREA NETWORK
OS/2
DOS
LAN MANAGER
WINDOWS NT
ECHO
PATHWORKS
ULTRIX
IPX/SPX
TCP/IP
NETWARE
OSF/1

ПОЛНЫЙ АССОРТИМЕНТ
ИЗДАВАЕМОЙ ИИД "ФИЛИНЪ"
ЛИТЕРАТУРЫ
/ПО ЦЕНАМ ИЗДАТЕЛЬСТВА/
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ПО СЛЕДУЮЩИМ АДРЕСАМ :

Москва:
Издательство "Филинъ"
125239, г. Москва, Новопетровская ул., д.1.
Телефон/факс: (095) 450 0041; 459 4131.

Санкт-Петербург:
Агентство "Бизнес-пресса" (НКФ "Инмар")
(официальный дилер)
191119, г. Санкт-Петербург, Разъезжая ул., д.39.
Телефон: (812) 164-7482, 164-5601
Факс: (812) 164-7482, 272-0561
E-Mail: root@inmar.spb.su

Новосибирск:
ЗАО "СибВерк"
(официальный дилер)
630058, г.Новосибирск, Русская ул., д.35.
Телефон: (383-2)35-50-49.

КНИГИ ИИД "ФИЛИНЪ"
ВЫ ТАКЖЕ МОЖЕТЕ КУПИТЬ:

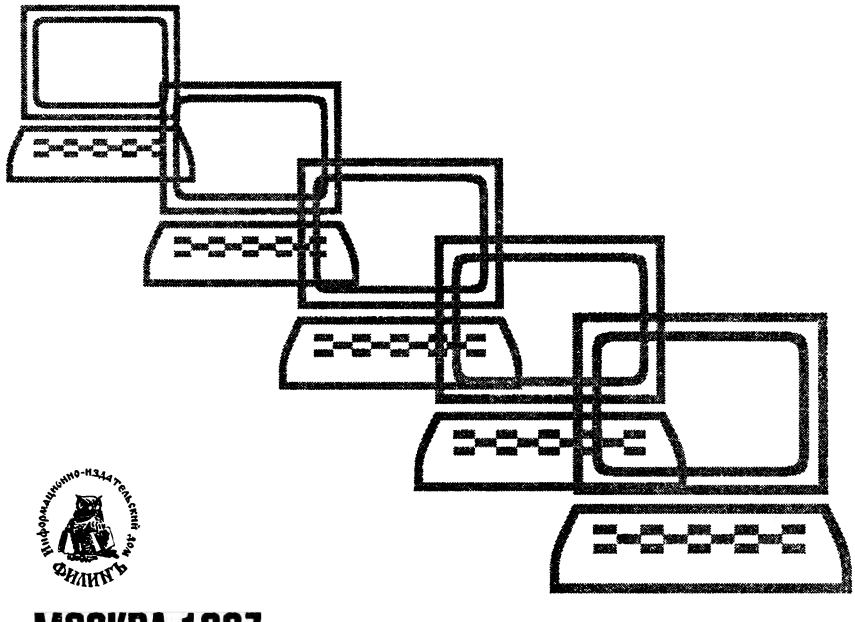
Нижний Новгород:
ТОО "МДЦ" отделение "Полезная книга"
Телефон: (831-2)67-04-74
Телефон/факс: (831-2)35-37-86

Client/Server Programming in PC LANs

Ed Barfield and Brian Walters

ЭД БАРФИЛД
БРАЙЕН УОЛТЕРС

ПРОГРАММИРОВАНИЕ "КЛИЕНТ-СЕРВЕР" В ЛОКАЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ СЕТЯХ



Информационно-издательский дом ФИЛИНЪ

МОСКВА 1997

УДК 681.324.066
ББК 32.973.202-018.2
Б 266

Б 266 Эд Барфилд, Брайен Уолтерс
Программирование «клиент-сервер» в локальных вычислительных сетях/ Перевод с англ. — М.: Информационно-издательский дом «Филинъ», 1997. — 424 с.

ISBN 5-89568-015-1
ISBN 1-878956-44-2 (англ.)

Эта книга является прежде всего учебником для разработки сетевых программ взаимодействия процессов в распределенной системе «клиент-сервер». Программирование в этой среде весьма осложнено наличием многочисленных сетевых операционных систем, каждая из которых имеет свой собственный сетевой интерфейс прикладного программирования.

Основное внимание сконцентрировано на программировании гнезд в средах DOS, Windows V3.1 и Windows NT — с помощью TCP/IP, а также в сети «клиент-сервер» NetWare, использующей IPX/SPX.

В книге представлен справочный материал — сетевые транспортные протоколы фирм Microsoft, Novell, DEC.

Для лучшего понимания этой книги необходимо хорошее знание языка программирования «Си» (C). Кроме того, полезен опыт работы в C++.

Книга может оказаться полезной и для администраторов локальных компьютерных сетей, различные элементы которых поставлены разными фирмами.

ББК 32.973.202-018.2

© ИИД «Филинъ», 1997

Copyright 1995 CMB Books, a division of Cardinal Business Media Inc.

© Перевод на русский язык ИИД «Филинъ», 1997

© Составление, оформление и репродуцируемый оригинал-макет ИИД «Филинъ», 1997

© Оформление обложки ИИД «Филинъ», 1997

© Оформление дискеты ИИД «Филинъ», 1997

ISBN 5-89568-015-1
ISBN 1-878956-44-2 (англ.)

Содержание

Введение 9
Рекомендуемая литература 15

ГЛАВА 1
Обзор сетевых операционных систем и средств передачи данных

Обзор сетевых ОС и средств передачи данных 17
ОС NetWare фирмы Novell 18
Протоколы NetWare 23
Адресация в ОС NetWare 28
LAN Manager 30
PATHWORKS для OpenVMS (LAN Manager) 33
Протоколы, интерфейсы прикладного программирования
и средства передачи данных сетевых операционных систем 35
NetBIOS 35
Протоколы TCP/IP 39
DECnet 50

ГЛАВА 2
Взаимодействие процессов: гнезда, вызовы удаленных процедур и именованные программные каналы

Методология взаимодействия процессов 55
Взаимодействие гнездовых процессов 57
Вызовы гнезд 67
Гнезда Windows 77
Программирование гнезд: Windows и DOS 78
Вызовы удаленных процедур 84
Именованные программные каналы 91

ГЛАВА 3
Гнезда TCP/IP в системе “клиент-сервер”: DOS и BSD UNIX

Введение 95
Клавиатурное “эхо” в системе “клиент-сервер” 95

ГЛАВА 4
Клиент и сервер в режиме "эхо": WINSOCK и Windows NT

Введение 115
Клиент в режиме "эхо" (с пояснениями) 117
Сервер в режиме "эхо" (с пояснениями) 126
USERLIB.C (с пояснениями) 133
Создание готовых к выполнению программ 139
Использование Visual C++ 139

ГЛАВА 5
Усовершенствованное программирование гнезд:
Мультирежим системы "клиент-сервер"

Введение 167
Работа с клиентами в мультирежиме (с пояснениями) 170
Работа с серверами в мультирежиме (с пояснениями) 176
USERLIB.C 186
Создание готовых к выполнению программ 187
Использование Visual C++ 187
Регистрация сервера (с пояснениями) 188
Регистрация Клиента 1 199
Регистрация Клиента 2 203
Регистрация Клиента 3 207
Регистрация Клиента 4 212
Регистрация Клиента 5 213

ГЛАВА 6
Библиотека гнезд C++

Введение 217
CSOCKET.H 219
CSOCKET.H (с пояснениями) 220
CSOCKET.CPP (с пояснениями) 226
Пример сервера 244
Пример сервера (с пояснениями) 245

ГЛАВА 7

Использование именованных программных каналов для сервера режима “эхо”

Введение 249
Именованные программные каналы клиента режима “эхо” (с пояснениями) 251
Сервер NPECHO (с пояснениями) 256
Построение программ в SDK 264
Использование языка Visual C++ для построения файлов 265
Примеры файлов регистрации 266

ГЛАВА 8

Программа многоклиентских программных именованных каналов

Введение 269
Изменения в коде программы сервера 270
Построение готовых к выполнению программ с помощью SDK 278
Построение готовых к выполнению программ с использованием языка Visual C++ 278
Примеры протоколов 279
Протокол Клиента 1 282
Протокол Клиента 2 282
Протокол Клиента 3 283
Протокол Клиента 4 283

ГЛАВА 9

Реализация программы на языке Visual C++

Введение 285
Цикл сервера 286
Функции гнезд 287
Создание проекта сервера 298
Модификация меню 301
Соединение пунктов меню с кодом программы 307

Описание класса CVisualDoc 313
VISUADOC.H (с пояснениями) 314
Реализация класса CVisualDoc 318
Новый VISUADOC.CPP (с пояснениями) 319
Создание SENDDIALOG 327
VISUADOC.CPP (с пояснениями) 332
Обновление интерфейса пользователя 339
Реализация функций UserLib (библиотеки пользователя). 349
Описание класса CVisualView 357
Новый VISUAVW.H 358
Реализация класса CVisualView 359
Новый VISUAVW.CPP 361
Первая попытка работы
в программе-компоновщике (BUILD) 363
Создание ловушки блокирования 364
Особенности, требующие внимательного отношения 365
Осуществление функционирования меню FdSets 366

ГЛАВА 10
Программирование гнезд и интерфейсов транспортного уровня NetWare

Интерфейс гнезд межсетевого/упорядоченного обмена
пакетами NetWare 367
Интерфейс транспортного уровня (TLI) 370
Библиотека TLI фирмы Novell 379
CSocket.cpp фирмы Novell (с пояснениями) 379
Пример сервера (с пояснениями) 401
Пример клиента фирмы Novell (с пояснениями) 403
Опции компоновки и компиляции фирмы Novell 405
Пример сеанса клиента 408
Пример сеанса сервера 411

АЛФАВИТНЫЙ УКАЗАТЕЛЬ 413

Введение

Описание

За последние годы применение компьютеров эволюционировало от централизованной интерактивной структуры на базе крупных компьютеров к распределенной архитектуре "клиент-сервер". Эта эволюция была осуществлена с помощью распространения локальных компьютерных сетей (Local Area Network — LAN) на базе персональных компьютеров.

Краеугольным камнем локальных компьютерных сетей является применение компьютеров в системе "клиент-сервер". Упрощенно такое применение компьютеров означает, что программа клиента в одной сетевой хостовой системе связывается с программой сервера в другой хостовой системе. Программа сервера стартует первой, пассивно ожидает запросов клиента и при их получении обрабатывает эти запросы с помощью различных механизмов.

Программирование в этой среде бывает весьма разнообразным вследствие наличия многочисленных сетевых операционных систем (Network Operating System — NOS), которые объединяют персональные компьютеры в локальные компьютерные сети. Каждая сетевая операционная система имеет свой собственный сетевой интерфейс прикладного программирования (Applications Programming Interface — API) на высоком уровне и различные поддерживаемые интерфейсы прикладного программирования более низкого уровня для осуществления связи. Больше возможностей (но и больше сложностей) для разработчика сетевого программного обеспечения предлагает интерфейс Windows фирмы Microsoft, с помощью которого осуществляется "ввод" этих программ клиента и сервера в среду DOS, OS/2, Windows for Workgroups или Windows NT. Таким образом внедряется другая методология программирования.

Эти программы клиента и сервера должны "говорить" на одном языке при передаче по сетевым каналам или "линиям", для того чтобы сообща выполнять полезные функции. Эти языки, или сетевые транспортные протоколы, включают в себя расширенный интерфейс пользователя для сетевой базовой системы ввода-вывода фирмы Microsoft (NetBIOS Extended User Interface — NetBEUI), межсетевой обмен пакетами/последовательный обмен пакетами (Internet Packet Exchange/ Sequenced Packet Exchange — IPX/SPX) фирмы Novell, глобальный протокол управления передачей (транспортный протокол)/межсетевой протокол (Transport Control Protocol/Internet Protocol — TCP/IP) UNIX и DECnet фирмы Digital Equipment Corporation. Транспортные сетевые средства действуют как носители

запросов интерфейсов прикладных программ высокого уровня по физическим линиям сети или, в последнее время, по радиоканалам.

Эта книга объясняет сложности, стоящие перед программистом, начинающим осуществлять программирование системы "клиент-сервер" в локальной компьютерной сети. В этом плане книга использует подход "задом наперед": сначала в ней представлены основы знаний в области сетевых протоколов и средств передачи, представленных в программах этой книги — TCP/IP и IPX фирмы Novell. На основе этих знаний исследуются интерфейсы прикладного программирования для взаимодействия процессов для программирования этих средств передачи — это включает в себя гнезда и именованные сетевые программные каналы (pipes). И, наконец, книга "поставляет конечный товар" в виде программ "клиент-сервер", выполненных в большинстве случаев с помощью визуальной модели Windows. В этой книге в качестве механизма кодирования, компиляции и компоновки программ использовался набор средств для разработки программного обеспечения Visual C++ фирмы Microsoft. Однако при внесении минимальных изменений в коды программ с большинством представленных программ должны работать и другие визуальные языковые компиляторы.

В этой книге представлен также обзор *вызовов удаленных процедур (Remote Procedure Calls)* — это другая методология взаимодействия процессов, однако в книге не приводится программа, использующая вызовы удаленных процедур.

Основное внимание в этой книге сконцентрировано на программировании гнезд — в средах DOS, Windows V3.1 и Windows NT — с помощью TCP/IP, а также в сети "клиент-сервер" NetWare, использующей IPX/SPX. В книге объясняется каждая типовая задача, которая должна решаться как клиентом, так и сервером, включая все соответствующие способы хронирования и синхронизации. Например, имеет смысл, чтобы одной из первых задач, решаемых клиентом в большинстве сред "клиент-сервер", было определение местоположения сервера в сети.

В этом разделе упор делается на описание того, как следует писать фрагменты программ для выполнения конкретной задачи в большинстве сред "клиент-сервер". Фрагменты кода программ должны быть компактными для каждой из сред, так как компактность очень важна при сетевом программировании взаимодействия процессов. Представлены протоколы выполнения программ для демонстрирования реальной работы программы.

Для реализации некоторых из этих задач и для демонстрирования того, как организовать модуль сетевого API, определяются классы C++ с соответствующими переопределенными функциями. Введение в описание языков C и C++ не приводится, и языковые конструкции подробно не обсуждаются. Например, в этой книге подробно не описывается, как работает оператор *teranary ?: operator* , как установить цикл *DO WHILE* или как *pointer dereference operator* -> используется для осуществления доступа к членам структуры. Упор в этой книге делается

на сетевое программирование взаимодействия процессов, а не на то, как программировать в С или С++. В каждой прикладной программе показывается полный код исходной программы наряду с описанием логического процесса, используемого для создания прикладной программы "с начальной позиции". Упоминаются необходимые файлы включения, а также библиотеки, с которыми вам необходимо осуществлять компоновку. Там, где это необходимо, приводятся формирующие файлы или сценарии команд для утилиты компиляции и компоновки NMAKE. Большинство описаний программ построены по принципу включения пояснений (аннотаций), когда за разделами кода программы, содержащими полную программу, как правило, следуют пояснения.

Примеры программ "клиент-сервер" демонстрируются в следующих областях:

- гнезда TCP/IP,
- гнезда SPX (Novell),
- гнезда Windows (WINSOCK)
- 32 именованных канала WIN.

В книге "Программирование в локальных компьютерных сетях" также обсуждаются основы переноса сетевых средств DOS/Windows, программ клиента/сервера в среду операционной системы новой технологии Microsoft — Microsoft New Technology — NT) и наоборот. С точки зрения перспективы клиента/сервера в DOS/Windows и Windows NT объясняется библиотечный интерфейс программирования WINSOCK.

Как пользоваться этой книгой

Хотя в этой книге и представлен определенный справочный материал, она является прежде всего учебником. Следует использовать ее в качестве отправной точки при написании сетевых программ взаимодействия процессов с интерфейсом Windows, который может постоянно находиться в большинстве сетевых операционных систем, например, LAN Manager, NetWare и PATHWORKS. Эта книга может также служить руководством по сетевому программированию для персонала, поддерживающего компьютерную сеть, различные элементы которой поставлены различными поставщиками.

"Программирование в локальных компьютерных сетях" подробно объясняет каждый аспект демонстрационных программ, а также ПОЧЕМУ принимаются те или иные подходы. В ней нет попыток объяснить все и каждую функцию интерфейса прикладного программирования, например программирования гнезд TCP. В ней объясняются только те функции, которые используются в примерах.

Предпосылки

Необходимым условием для чтения этой книги является хорошее знание языка программирования "Си" (С). Предполагается, хотя и не требуется, что читатель знаком с программированием в среде Windows и хорошо разбирается в сетевых

системах в смысле модели взаимосвязи открытых систем (OSI). Кроме того, полезен опыт работы в C++ для понимания демонстрируемых классов C++. Однако для выполнения этого предварительного условия вы можете взять учебник по C++ фирмы Microsoft с комплектом документации Visual C++ или любой другой учебник.

ПРИМЕЧАНИЕ: Прикладная программа Microsoft на Visual C++, версия 1.10 (в NT) и версия 1.0 (в DOS) была использована для компиляции и компоновки большинства примеров программирования, приведенных в этой книге. Однако версия 1.5 языка Visual C++, как и более поздние версии, также должна работать. Другие компиляторы языка C++, такие как Borland, могут быть использованы для большинства программ, за исключением программы на Visual C++, приведенной в главе 9. Это зависит от базовых классов Microsoft (Microsoft Foundation Classes).

Структура книги

Книга состоит из следующих глав:

- **Глава 1: Обзор сетевых операционных систем и средств передачи данных (транспортных средств)** — В этой главе рассматриваются три сетевые операционные системы: NetWare, LAN Manager фирмы Microsoft и PATHWORKS фирмы Digital Equipment Corporation. Упор сделан на представлении обзора набора протоколов TCP/IP и IPX/SPX и той роли, которую они играют в этих сетевых операционных системах. Эти протоколы используются в примерах программирования, представленных в этой книге.

- **Глава 2: Взаимодействие процессов: гнезда, вызовы удаленных процедур и именованные программные каналы** — В этой главе исследуется внутренний механизм передачи данных интерфейса прикладных программ этих сетевых операционных систем — взаимодействие между процессами. Изучаемые методы взаимодействия между процессами — это гнезда, вызовы удаленных процедур и именованные программные каналы (pipe). Упор сделан на гнездах, которые также лежат в основе связи "клиент-сервер" в NetWare, а также на вызовах удаленных процедур при использовании в TCP/IP.

- **Глава 3: Гнезда TCP/IP клиента-сервера: DOS и BSD UNIX** — В этой главе демонстрируется простая программа гнезд TCP/IP системы "клиент-сервер" от клиента DOS PATHWORKS к серверу ULTRIX или OSF/1. Подробно объясняются разделы кода программы, а также соображения переноса программы при ее переносе для клиента PATHWORKS V5. Код программы клиента переносится для клиента DOS TCP/IP.

- **Глава 4: Клиент и сервер в режиме "эхо" (ECHO): WINSOCK и Windows NT** — В этой главе подробно объясняется реализация эхо между клиентом и сервером с помощью интерфейса WINSOCK в системе Windows NT.

Описывается "построение" проекта под Visual C++. Представлены файлы регистрации взаимодействия "клиент-сервер".

- **Глава 5: Усовершенствованное программирование гнезд: работа сервера в мультирежиме** — Более сложная реализация режима, описанного в главе 4. Показана работа в режиме, когда сервер может обслуживать нескольких клиентов, осуществляя связь и отсылая данные через гнезда TCP. Демонстрируются построение управляющей системы, а также образцы файлов регистрации взаимодействия сеансов связи с несколькими клиентами.

- **Глава 6: Библиотека гнезд в C++** — Представлена библиотека класса гнезд C++ наряду с программой, демонстрирующей применение библиотеки классов. В этой главе устанавливается этап для "полного соответствия C++" с помощью реализации библиотеки классов.

- **Глава 7: Использование именованных каналов для сервера в режиме "эхо" (ECHO)** — Представлен пример программирования именованных программных каналов (pipes) NT, а также информация о том, как построить программу. В этой главе представлены файлы регистрации сеанса клиента с сервером с именованными программными каналами и объяснены все вызовы именованных программных каналов с моделью пояснений, описанной в предыдущей главе.

- **Глава 8: Программа именованных программных каналов при работе с несколькими клиентами** — Подобно тому как в главе 5 представлена работа системы "клиент-сервер" в мультирежиме, эта глава демонстрирует систему "клиент-сервер" с несколькими именованными программными каналами. Дается описание отличия от модели "клиент-сервер с одним именованным каналом" и приводятся файлы регистрации.

- **Глава 9: Реализация программы с Visual C++** — В этой главе с помощью базовых классов Microsoft и Visual C++ реализуется полный графический интерфейс с программой гнезд сервера. Для регистрации используется зона клиента в прикладной программе, а система меню динамически изменяется для указания активных гнезд.

- **Глава 10: Программирование гнезд NetWare для интерфейса транспортного уровня** — В этой главе в общем плане объясняется интерфейс программирования гнезд NetWare, затем демонстрируется NetWare SPX, гнездовая программа на основе логических соединений, использующая интерфейс вызова TLI (Transport Layer Interface — интерфейс транспортного уровня).

Условные обозначения

Приведенная ниже таблица описывает условия применения шрифтов в книге.

Условные обозначения	Описание
курсив	Используется для указания имени вызова функции интерфейса прикладного программирования и ее аргумента (аргументов), а также большинства программных переменных. Имя аргумента должно быть заменено действительным значением этого аргумента. Курсивный текст также используется для определяемых слов
жирный шрифт	Используется для ключевых слов, таких, как структуры и их поля. Запись должна точно соответствовать их написанию — так, как они должны появляться в исходной программе. Кроме того, этот тип шрифта используется для смыслового подчеркивания в тех случаях, когда этого требуют определенные компоненты вызова интерфейса прикладного программирования или при выборе меню
Программный шрифт	Используется для программ, отрывков программ в тексте и элементов меню из экранов Windows
Жирный программный шрифт	Используется для выделения кода в кодовом листинге
ЗАГЛАВНЫЕ БУКВЫ	Используются для наименования файлов, путей доступа, наименований структур, констант и некоторых команд (таких, как команды NetBIOS), с тем исключением, что при написании некоторых путей доступа используется нижний регистр из-за условностей операционной системы
Угловые скобки <...>	Используются для обозначения клавиатурных комбинаций, таких, как: <Ctrl><Alt><Del>
Квадратные скобки [...]	Используются для обозначения дополнительных (опциональных) параметров

Рекомендуемая литература

Barfield, Edward. *PATHWORKS V5 Network Administration Guide.* (CBM Books, 1995) ISBN 1-878956-49-3.

CAE Specification: Protocols for X/Open PC Interworking: SMB, Version 2. (X/Open Company Ltd., U.K., September, 1992). ISBN: 1-872630-45 6.

Comer, Douglas E. *Internetworking With TCP/IP, Volume I.* (Prentice-Hall, Inc., 1991). ISBN: 0-13-468505-9.

Heywood, Drew, Janos (John) Jerney, Jon Johnston, et al. *LAN Connectivity.* (New Riders Publishing, 1992). ISBN: 1-56205-048-6.

Microsoft Technical Resource Group. *Microsoft LAN Manager Resource Kit.* (Microsoft Corporation, 1992). Document No.: 096-044-552. Part Number: 100-000905-001.

Microsoft Visual C++ Programmer's Guides: C++ Tutorial, Class Library User's Guide and Programming Techniques. (Three books combined into one volume: Microsoft Corporation, 1993). Document No.: DB29682-0193.

NetWare Client Transport Protocol API for C. (Novell, Inc., 1993). Part Number: 100-001859-001.

NetWare Connection. "NetWare V4.x: Questions and Answers". (NetWare Users International, November/December, 1993).

NetWare Programmer's Guide for C. (Novell, Inc., 1993). Part Number: 100-001856-001.

Open Software Foundation. *OSF/1 Operating System: Network Applications Programmer's Guide.* (Prentice-Hall, Inc., 1992). ISBN: 0-13-640145-7.

PATHWORKS Network Programming Reference. (Digital Equipment Corporation, 1994). Order Number: AA-PAFJD-TK.

PATHWORKS for OS/2: Microsoft LAN Manager Programmer's Reference. (Microsoft Press, 1990). Part Number: AA-PB9CC-TK (Distributed by Digital Equipment Corporation).

PATHWORKS V5: Advanced LAN Management/Support for the OpenVMS Server Student Guide. (Digital Equipment Corporation,1994). Course Code:EY-M811E-SG-0001.

Stevens, Richard V. *UNIX Network Programming.* (McGraw-Hill, 1990). ISBN: 0-13-949876-1.

Глава 1

Обзор сетевых операционных систем и средств передачи данных

ОБЗОР СЕТЕВЫХ ОПЕРАЦИОННЫХ СИСТЕМ И СРЕДСТВ ПЕРЕДАЧИ ДАННЫХ

Введение

Основой архитектуры компьютерных локальных сетей является применение компьютеров в системе "клиент-сервер". Термин *распределенное применение компьютеров* (*distributed computing*) означает распределение вычислительных задач от централизованных, крупных вычислительных машин к меньшим по размеру рабочим станциям в коллективных компьютерных локальных сетях. Вследствие того что стержнем архитектуры компьютерных локальных сетей является связь между сервером и клиентом, возник термин *распределенное применение компьютеров в системе "клиент-сервер"* (*distribute client/server computing*). Содержанием настоящей книги являются основы программирования в этой распределенной среде "клиент-сервер".

За последние годы некоторые поставщики компьютеров написали программное обеспечение, которое связывает компьютеры, работающие в сети, в связные структуры. Эти сетевые операционные системы (NOS) объединяют код, который относится к семи уровням модели взаимосвязи открытых систем (OSI). Модель OSI представляет собой стандартизованную модель связи, созданную Международной организацией по стандартизации. Поставщики средств, которые придерживаются условий этой модели, называются "соответствующими открытым системам" ("open-system compliant").

Сетевые операционные системы включают в себя *файловый сервер* (*file server*), который обеспечивает единый пункт управления инсталляцией прикладной программы и хранения данных, и поддерживаемую сетевую систему передачи данных, которая несет данные для прикладных программ в системе "клиент-сервер".

В этой главе мы рассмотрим следующие сетевые операционные системы: LAN Manager фирмы Microsoft, PATHWORKS фирмы DEC и NetWare фирмы Novell. Упор будет сделан на внутренней сетевой архитектуре каждой из этих сетевых операционных систем — особенно на наборе протоколов TCP/IP и IPX, на которых мы будем демонстрировать программы системы "клиент-сервер".

Операционная система NetWare фирмы Novell

Описание

В настоящее время наиболее распространенной сетевой операционной системой является *NetWare* фирмы Novell Inc. Это сетевая операционная система, которая работает, как правило, с компьютерами с процессором 386 и выше в системе "клиент-сервер" (отметим, что версии с номером ниже 3.11 можно устанавливать на компьютерных системах Intel 286). Последние версии NetWare — это версии 4.x, где x — это номер, представляющий обновленную версию по сравнению с версией 4.0. Версия 4.0 была выпущена в мае 1993 года. Версия 3.12 является обновлением версии 3.11 и предлагает опцию замены у клиента оболочки NetWare виртуальными загружаемыми модулями (Virtual Loadable Modules).

Программное обеспечение сервера NetWare устанавливается на персональных компьютерах с процессором 386 или выше, в то время как персональные компьютеры клиентов конфигурируются с *оболочкой* NetWare, которая позволяет им соединяться с сервером и/или входить в систему и использовать имеющиеся виды сервиса. Виды сервиса — это такие возможности, как совместное использование файлов и принтеров, передача сообщений по электронной почте, доступ к базе данных и к другим прикладным программам. Наиболее фундаментальным видом сервиса является *каталог* (*directory*) файловой системы сервера, к которому рабочая станция подключается для доступа с помощью символа дискового драйва местной DOS. Этот вид сервиса называется просто *служба каталога* (*directory service*).

Набор базовых протоколов NetWare: IXP/SPX

Платформой, на которой строятся соединения в системе клиент-сервер в NetWare, является набор протоколов IPX/SPX, написанный фирмой Novell. Фирма Novell адаптировала архитектуру IPX/SPX из набора протоколов Xerox Network Systems (XNS) и модифицировала и усовершенствовала этот набор для приспособления его к требованиям NetWare.

Связь клиента с сервером в NetWare: драйвер IPX

Internetwork Packet eXchange (IPX) — *межсетевой пакетный обмен* — это протокол низкого уровня, реализованный в архитектуре NetWare фирмы Novell. IPX.COM — драйвер, созданный как часть инсталляции рабочей станции клиента NetWare. Это обеспечивает персональному компьютеру доступ к серверу NetWare в сети.

Драйвер IPX.COM — это все, что требуется для клиента NetWare для программирования SPX, программы "гнездо-гнездо", при связи с другим клиентом, который в этом случае действует как сервер.

Оболочка NetWare

Оболочка NetWare — это резидентная программа "завершить и остаться в памяти" (TSR — Terminate-and-Stay-Resident), которая определяет, будет ли запрос на применение обрабатываться операционной системой DOS или же программным обеспечением NetWare. В последнем случае оболочка инициирует пакет протоколов

(IPX/SPX), необходимый для обмена данными с сервером NetWare. NET*n*.COM, где *n* — 3, 4, 5 или 6 в зависимости от версии DOS, или NETX.COM — действительное имя файла — копируется в устройство самозагрузки персонального компьютера в качестве части процесса инсталляции и загружается, когда требуется доступ к серверу NetWare. NET*n*.COM — это программное обеспечение, которое включает в себя оболочку NetWare. Рабочая станция с сервисом оболочки дает на сервер запрос, передавая эти пакеты запроса на сервер через IPX.COM и обрабатывая ответные пакеты сервера. Она выполняет эти функции управлением работой гнезд. Как только оболочка загружена, она автоматически предпринимает попытки осуществить связь с одним из серверов сети.

Архитектура сервера NetWare

Сервер 386 версии 3.11 NetWare предусматривает загружаемые модули NetWare (NLM) и поддержку опционального *пространства имен* (*name space*) (идентификация системы файлов) для расширения сетевой среды. Загружаемые модули NLM — это программные модули, которые могут динамически загружаться или разгружаться из сервера при его работе.

На рис. 1-1 показана обобщенная версия серверной архитектуры NetWare.

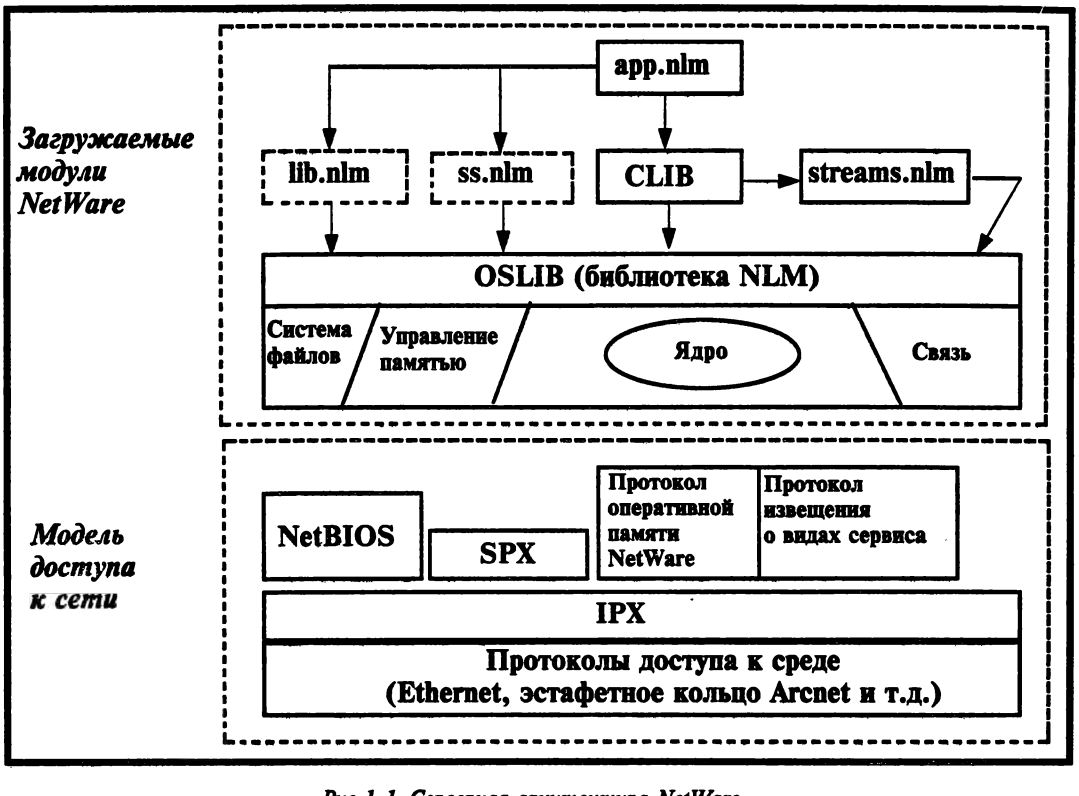


Рис.1-1. Серверная архитектура NetWare.

На этом рисунке к архитектуре NLM принадлежат следующие элементы:

- OSLIB — библиотека интерфейсов низкого уровня, которая переносит многие вызываемые стандартные программы в прикладные программы NLM. Она включает в себя интерфейс прикладного программирования уровня поддержки канала (Link Support Layer API) для драйвера локальной сети и модулей NLM стека протоколов, интерфейс прикладного программирования дисковой подсистемы для модулей NLM дискового драйвера, интерфейс прикладного программирования системы файлов для модулей NLM пространства имен, а также многие другие функции и переменные для всех модулей NLM. Это — интерфейс с механизмом вызова гнезд.
- Некоторые модули NLM, такие, как утилиты MONITOR и INSTALL фирмы NetWare, имеют непосредственный доступ к OSLIB. Это имеет место потому, что эти утилиты требуют доступа к важным внутренним элементам операционной системы, и отчасти также потому, что они были написаны до появления на свет CLIB — библиотечного интерфейса языка С.
- Другие модули NLM, такие, как модуль STREAMS, также имеют непосредственный доступ в OSLIB. Библиотечные модули NLM в NetWare образуют второй интерфейс прикладного программирования на OSLIB, к которому имеет доступ большинство модулей NLM третьей стороны.
- CLIB является краеугольным камнем вторичного интерфейса прикладного программирования. Для функций языка С, таких, как {длина цепочки} *strlen()*, CLIB сам выполняет работу и возвращает ответ на запрашивающий модуль NLM. Для выполнения таких функций, как *read()*, CLIB запрашивает помощь у OSLIB.
- *lib.nlm* представляет собой ряд библиотечных модулей NLM NetWare.
- *ss.nlm* представляет собой значительно меньший библиотечный модуль NLM, описанный в документации комплекта распространения программного обеспечения фирмы Novell (Novell Software Distribution Kit).

Ниже приводятся факты, относящиеся к *модели доступа к сети* (*network access model*) в диаграмме серверной архитектуры NetWare:

- IPX представляет собой дейтаграмму, протокол без установления логического соединения, аналогичный IP в наборе TCP/IP, в том смысле, что он не требует подтверждения приема (квитирования) для каждого отосланного пакета. Протоколы выше IPX должны обеспечивать управление логическим соединением.
- Протокол маршрутной информации (Routing Information Protocol — RIP) — облегчает обмен маршрутной информацией в межсетевой работе NetWare. RIP является производной от набора протоколов сетевых систем Xerox — Xerox Network Systems — XNS (см. соответствующие материалы).
- Протокол извещения о видах сервиса (Service Advertising Protocol — SAP) обеспечивает работу сервисных узлов — файловых серверов, принтерных

серверов и шлюзовых серверов — для извещения о их видах сервиса и адресах. Через информационный сервисный протокол клиенты, посылая пакеты запросов, могут определить, какие виды сервиса доступны в сети, и получают межсетевой адрес узлов, или серверов, где они могут получить доступ к этим видам сервиса.

Версия 4.0 NetWare

Версия 4.x NetWare существенно отличается от версии 3.11 своей характеристикой видов сервиса каталога (Directory Services). NetWare V.4x рассматривает мультисерверные сети как единую логическую сеть, аналогично тому, как реализованы домены в LAN Manager, если бы вы добавили некоторые поисковые виды сервиса Желтых страниц (YP) или какой-либо другой сервис просмотра имен.

ПРИМЕЧАНИЕ: Желтые страницы — это распределенный сервис просмотра данных для совместного использования информации системами в сети с протоколами IP. Информация, распространяемая Желтыми страницами, хранится в *картах* (*maps*) или же наборах файлов базы данных. Именованный набор карт Желтых страниц известен под названием *домена* (*domain*). Желтые страницы основаны на модели "клиент-сервер".

При работе с NetWare 4.x пользователи входят в систему один раз и имеют доступ к тем ресурсам сети, на которые они имеют право — независимо от того, где находятся эти ресурсы. Эти ресурсы могут находиться в любом сервере сети, к которой имеется доступ.

Сервис Каталога NetWare (NetWare Directory Services) является инструментом версии 4.0, который обеспечивает возможность логического поиска по каталогу. Мультисерверные сети фирмы Novell в настоящее время организованы как единое целое. Новые виды сервиса Каталога версии 4.0 фирмы Novell, основанные на X.500, упрощают повседневную работу пользователей сети NetWare. X.500 имеет отношение к рекомендации Международного консультативного комитета по телеграфии и телефонии (CCITT) относительно стандартизованных возможностей каталога, которые требуются применениями взаимодействия открытых систем (OSI), процессами управления и другими элементами уровней OSI. Одними из таких возможностей являются система присвоения имен и отображение имени в адрес, дружественные по отношению к пользователю.

Directory Services управляет всеми ресурсами сети, такими, как пользователи, принтеры и серверы, с помощью доступной для всех базы данных. Все ресурсы сети называются *объектами* (*objects*). Например, пользовательский объект представляет собой пользователя, а принтерный объект — принтер. Объект включает в себя не только сам ресурс, но и информацию, связанную с этим ресурсом.

Версия NetWare 3.11, как и 3.12, по существу, управляется *сетевой системной базой данных* (bindery), которая представляет собой базу данных, описывающую

сервер. Сетевая системная база данных включает в себя объекты, например, *свойства* (*properties*) и *наборы данных свойств* (*property data sets*). Свойства — это характеристики объектов, а наборы данных свойств — это величины, присвоенные конкретным характеристикам. Примером объекта может служить имя пользователя, примером свойства может служить пароль, примером набора данных свойств могут служить действительные символьные данные, включающие в себя пароль пользователя.

Сетевая системная база данных строится как база данных из плоских файлов. Это создает определенные трудности, ытак как когда сетевая системная база данных содержит большое количество объектов, для нахождения данного объекта должна сканироваться вся сетевая системная база данных. Однако Directory Services имеет иерархическую структуру и содержит дополнительные типы объектов. Таким образом, все сетевые ресурсы могут быть включены в дерево каталога, что облегчает управление и поиск.

Схема каталога (*Directory Schema*) является ключом к организации базы данных V4.x. Она представляет собой определение типов информации, хранящейся в базе данных каталога. Она диктует требования, пределы и отношения объектов, которые могут создаваться и находиться в каталоге.

Эта схема содержит три основных компонента:

- классы объектов,
- типы атрибутов,
- синтаксисы атрибутов.

Объекты идентифицируются с помощью их собственных имен, а также с помощью имени *контейнера* (*container*) или *родительских* (*parent*) объектов. Собственное имя объекта называется *относительным отличительным именем* (*Relative Distinguished Name — RDN*). Полное имя объекта (включая родительское) называется полным именем, или отличительным именем (Distinguished Name — DN).

Например, частичное имя человека может быть таким:

CN=EDWARD

Полное имя объекта может быть таким:

CN=BOB.OU=WRITERS.O=PUBLISHER.C=US

Эта цепочка определяет пользователя через общее имя, или (CN) BOB, в организационной единице (OU) WRITERS в стране (C) Соединенные Штаты.

Определение класса объектов, организация и соответствущие структурные правила становятся мощными организационными инструментами и делают серверную сеть NetWare доступной в глобальном масштабе и легко управляемой. Дальней-

шее исследование версии 4.x NetWare в этой книге не осуществляется, так как примеры программирования NetWare, приведенные далее в этой книге, являются примерами гнезд IPX низкого уровня. Последние SDK фирмы Netware для клиента включают в себя все запросы, относящиеся к Directory Services, а также вызовам гнезд IPX.

Протоколы NetWare

IPX/SPX, XNS и модель взаимосвязи открытых систем (OSI)

Поскольку IPX/SPX фирмы Novell были выведены из сетевых систем Xerox (XNS), целесообразно дать более подробный анализ того, как набор протоколов XNS образует основу для протоколов IPX/SPX NetWare. В этом обсуждении мы также покажем наложение наборов протоколов XNS и NetWare на эталонную модель взаимосвязи открытых систем (OSI), с тем чтобы сравнить два набора протоколов.

Сетевые системы Xerox (XNS)

Термин *XNS* (*Xerox Network Systems*) коллективно относится к набору межсетевых протоколов, разработанных фирмой Xerox Corporation в конце 70-х годов. Набор протоколов аналогичен протоколам TCP/IP, однако, XNS использует другие форматы пакетов и другую терминологию.

Протоколы XNS

Следующие протоколы и их описания включают в себя набор протоколов XNS:

- Эхо-протокол (ECHO) — заставляет главный компьютер (хост) дублировать принимаемый пакет.
- Протокол маршрутной информации (Routing Information Protocol — RIP) — используется для поддержки маршрутной базы данных для использования на главном компьютере (хосте) при направлении пакетов межсетевого протокола данных (Internet Data Protocol — IDP) на другой хост. Обычно процесс задания маршрута имеет место на главном процессоре, и этот процесс использует RIP для поддержки базы данных.
- Протокол пакетного обмена (Packet Exchange Protocol — PEX) — ненадежный, без установления логического соединения (*connectionless*), дейтаграммный (*datagram*) протокол для процессов пользователя. Протокол без логического соединения — это такой протокол, в котором нет конечных точек логического соединения, таких, которые имеются в соединениях от гнезда к гнезду в протоколе управления передачей (TCP). Протоколы без логического соединения могут передаваться в широковещательном режиме (broadcast) или для нескольких (multicast) станций сети. Дейтаграмма представляет собой основную единицу информации, проходящей через сеть XNS или TCP/IP. PEX не является надежным протоколом, что означает, что не гарантируется передача с помощью пакетов квитирования. Он ретранслирует, но не выполняет дублирующие функции.

- Протокол последовательной передачи пакетов (Sequenced Packet Protocol — SPP) — ориентированный на логическое соединение надежный протокол для процессов пользователя. Он обеспечивает поток байтов для процесса пользователя с опциональными границами сообщений. Это наиболее часто используемый протокол в наборе XNS. Он аналогичен TCP в наборе протоколов Internet.
- Протокол ошибок (ERROR) — может быть использован любым процессом для сообщения об обнаружении ошибки и, следовательно, об отмене пакета.
- Протокол межсетевой дейтаграммы (Internet Datagram Protocol — IDP) — ненадежный, без логического соединения, протокол дейтаграммы, который обеспечивает сервис доставки пакета для всех вышеуказанных протоколов.

Организация уровней в наборе протоколов XNS

На рис. 1-2 показано наложение набора протоколов XNS на модель взаимодействия открытых систем.

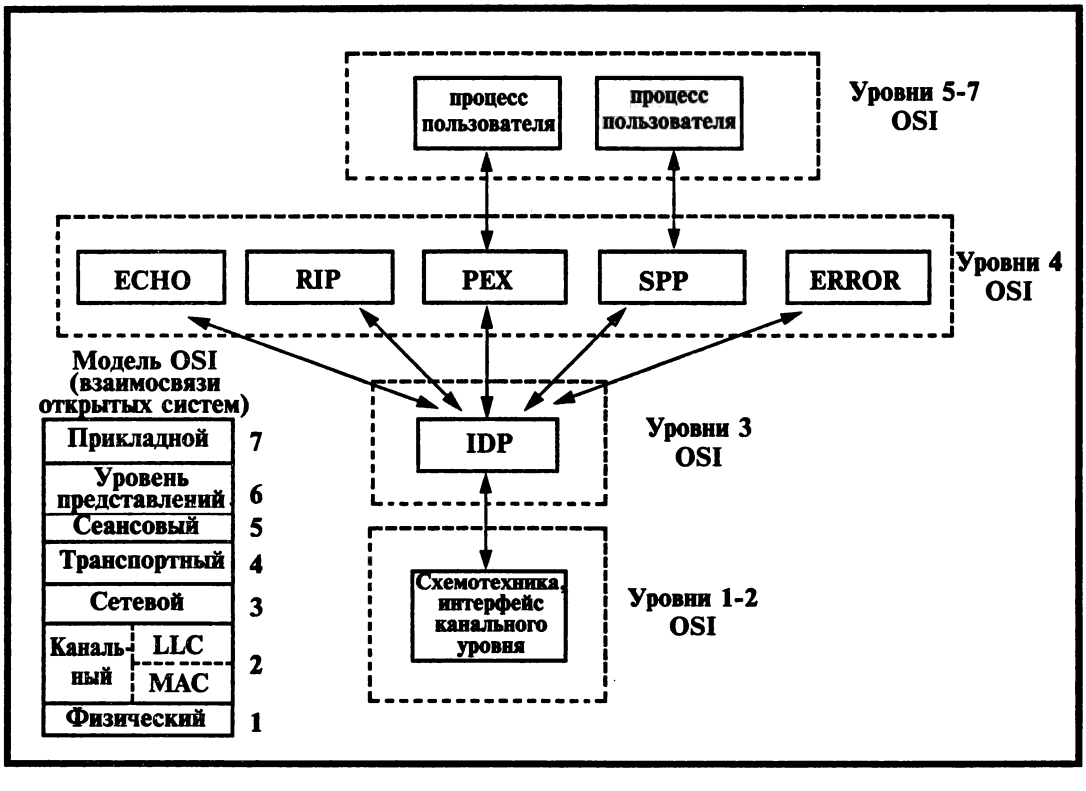


Рис.1-2. Набор протоколов XNS относительно модели OSI.

Протоколы NetWare

Протоколы NetWare определяют пакетные форматы, используемые для обмена информацией с сетевой средой NetWare. В принципе, имеется семь различных протоколов, используемых для связи в рамках собственной среды NetWare. Большинство из них аналогичны набору XNS. Они включают в себя следующее:

- Управление доступом к носителю данных (Medium Access Control — MAC).

  ПРИМЕЧАНИЕ: Этот протокол определяет способ, которым многоканальные интерфейсные драйверы (Multiple Link Interface Drivers — MLID) имеют доступ к устройствам управления сети при использовании в контексте открытого интерфейса канальных данных (Open Data Link Interface — ODI) фирмы Novell. В действительности термин "управление доступом к носителю данных" связан с подуровнем MAC канального уровня OSI. Уровень MAC и подуровень управления канальным уровнем (Link Layer Control — LLC) над ним включают в себя канальный уровень,
- Межсетевой обмен пакетами (Internet Packet Exchange — IPX),
- Последовательный обмен пакетами (Sequenced Packet Exchange — SPX),
- Протокол маршрутной информации (Routing Information Protocol — RIP),
- Протокол извещения о видах сервиса (Service Advertising Protocol — SAP),
- Протокол ядра NetWare (NetWare Core Protocol — NCP) — NetBIOS.

Ниже дается обзор каждого из этих протоколов.

Управление доступом к носителю данных

В компьютерной среде DOS драйвер доступа к носителю данных (MAC) является драйвером устройства, которое реализует подуровень MAC канального уровня (Link Layer) для конкретной спецификации канального уровня, такой, как интерфейсные спецификации сетевого драйвера фирмы Майкрософт (Microsoft's Network Driver Interface Specifications — NDIS). Драйверы MAC управляют адаптерной платой сети, соединенной с шиной персонального компьютера или же с линиями передачи данных на блок центрального процессора. Обычно они загружаются от CONFIG.SYS для реализации NDIS или с командного файла для ODI фирмы NetWare. Исключением является среда версии 5.x PATHWORKS фирмы DEC, в которой драйверы MAC загружаются с помощью другого загрузчика.

Межсетевой обмен пакетами (IPX)

Протокол межсетевого обмена пакетами (IPX) был принят исходя из IDP — межсетевого дейтаграммного протокола Xerox Network System. IPX представляет собой протокол без логического соединения и не требует квитирования для каждого посланного пакета. Это подтверждение приема пакета, или управление соединением, должно быть обеспечено протоколами выше IPX. IPX определяет межсетевые и межузловые схемы адресации, опираясь на схемотехнику сети для определения

узловой адресации. Определение маршрутов является первичной функцией IPX. IPX принимает маршрутные решения на основе информации о доступности сети, предоставляемой протоколом маршрутной информации (RIP).

Последовательный обмен пакетами (SPX)

SPX — это интерфейс транспортного уровня фирмы Novell, который обеспечивает передачу пакета, ориентированного на логическое соединение. Разработанный на основе протокола последовательной передачи пакетов (SPP) фирмы Xerox, SPX увеличивает возможности протокола IPX обеспечением надежной передачи данных. Одной широкоприменяемой реализацией SPX является утилита удаленной консоли (Remote Console — RCONSOLE) фирмы NetWare. Эта утилита позволяет рабочей станции, не являющейся пультом (консолью) сервера, действовать так же, как и пульт сервера.

Протокол маршрутной информации (RIP)

Протокол маршрутной информации облегчает обмен маршрутной информацией в объединенной сети NetWare. Как и IPX, RIP был выведен из XNS. Однако в пакетную структуру было введено изменение для улучшения критерия принятия решений относительно самого быстрого пути к месту назначения. Это изменение запрещает интеграцию типа “один к одному” протокола RIP фирмы NetWare с другими реализациями XNS.

Протокол извещения о видах сервиса (SAP)

Протокол извещения (объявления) (Service Advertising Protocol — SAP) о видах сервиса представляет средство для узлов обеспечения сервиса, таких, как файл-серверы и принт-серверы, для объявления их видов сервиса и адресов. Он имеет свое происхождение от определения "*клирингхаус*" (*Clearinghouse*) в Xerox Network System. Клирингхаус — это вид сервиса, используемый для определения местоположения именованного объекта и превращения имени в адрес. В качестве своей основы функция “клирингхаус” использует вызов удаленной процедуры (Remote Procedure Call — RPC). Протокол объявления видов сервиса является адаптацией, которая делает процесс добавления и снятия видов сервиса в сети более динамичным.

Через этот протокол клиенты в сети могут определить, какие виды сервиса в сети доступны в настоящий момент, и получить межсетевой адрес узлов или серверов, от которых они могут получить доступ к этим видам сервиса. Рабочая станция должна иметь адрес сервера до начала сеанса с ним. Как и RIP, протокол объявления видов сервиса использует для передачи данных IPX и MAC.

Протокол ядра NetWare (NCP)

Протокол ядра NetWare (NetWare Core Protocol — NCP) делает возможным взаимодействие между клиентами и файлсерверами Он определяет два аспекта взаимодействия между клиентами и серверами — управление связью и кодирование

запросов на сервис. Создание и обработка пакетов NetWare осуществляется оболочкой NetWare или устройством запросов OS/2.

NCP определяет свой собственный протокол для управления сеансом и контроля ошибок на уровне пакетов. Когда клиент устанавливает сеанс работы с файлсервером, ему присваивается номер присоединения. Этот номер должен включаться во все последующие запросы на сервис со стороны клиента. Таким образом файлсервер может проследить, какие клиенты осуществляют запросы.

Каждому пакету запросов NCP, пересылаемому по данному соединению, оболочка клиента присваивает последовательный номер. Первому запросу, следующему за установлением соединения, присваивается номер 1, и этот номер увеличивается на единицу для каждого последующего запроса. Когда файлсервер заканчивает обработку запроса, он помещает последовательный номер для этого запроса в ответный пакет. Таким образом, клиент может быть уверен, что он получает правильный ответ на каждый из запросов, которые он посылает.

Каждому виду сервиса, доступному файлсерверу, присваивается номер. Когда оболочка рабочей станции передает запрос на сервер, она помещает этот номер в кодовое поле сервиса пакета NCP.

Сетевая базовая система ввода/вывода (NetBIOS)

Интерфейс сеансового уровня NetBIOS фактически стал промышленным стандартом. NetWare обеспечивает эмуляционный пакет NetBIOS, который работает поверх протокола пакетного обмена (Packet Exchange Protocol — PEP). PEP обеспечивает другой транспортный уровень, принятый фирмой Novell из набора XNS — из PEX, который представляет собой протокол пакетного обмена XNS.

После обработки вызовов NetBIOS PEP посылает их на IPX. Эмуляционный пакет программ NetBIOS фирмы Novell поддерживает все функциональные вызовы NetBIOS. Эмуляция NetBIOS позволяет программам, написанным в соответствии с требованиями NetBIOS, работать в системе NetWare.

Протоколы NetWare, отображенные в OSI

Отображение NetWare в OSI является непростым делом вследствие многочисленных протоколов и прикладных программных интерфейсов (API), связанных с Netware. NetWare поддерживает множество стандартов компоновочного уровня LAN — Ethernet II, IEEE 802.3 (метод доступа Ethernet CSMA/CD), IEEE 802.2 SNAP, IEEE 802.5 (Token Ring — эстафетное кольцо), эстафетное кольцо IBM и ARCNET. Функциональность сетевого или маршрутного уровня управляется межсетевым обменом пакетами (IPX), в то время как последовательный обмен пакетами (SPX) обеспечивает интерфейс транспортного уровня. Сервис сеансового, транспортного и сетевого уровней обеспечивается пакетом программ NetBIOS фирмы Novell.

Оболочка NetWare обеспечивает интерфейс между вызовами операционной системы и сетевым сервисом нижнего уровня. Например, оболочка "ловит" ("hooks"), или принимает на себя, прерывание "зонтика" ("umbrella") DOS, или INT 21H. Это прерывание является механизмом, с помощью которого прикладные программы DOS осуществляют запросы в операционную систему. Интерфейс оболочки со стандартной программой INT 21H определяет, предназначается ли ввод/вывод для драйвера NetWare, который был занесен в каталог сетевого сервера, или же для принтсервера NetWare.

Протоколы ядра NetWare (NCP) обеспечивают интерфейс уровня приложений с сервисом системы файлов NetWare. На рис. 1-3 приведена диаграмма отображения протоколов NetWare в OSI.

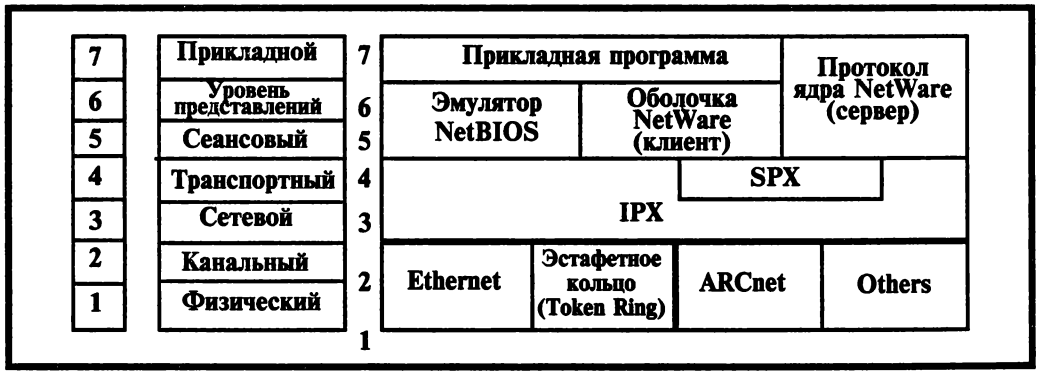


Рис. 1-3. Протоколы NetWare в модели OSI.

ПРИМЕЧАНИЕ: IEEE — Институт инженеров по электротехнике и радиоэлектронике, организация по стандартам, которая разработала стандарты канального уровня для передачи компьютерных кадров по целому ряду физических сред, таких как Ethernet и Token Ring.

Адресация в системе NetWare

Адреса XNS

Так как набор протоколов NetWare был разработан после разработки набора протоколов XNS, было сочтено целесообразным сделать адрес NetWare подобным адресу XNS.

Адрес XNS, описывающий соединение между двумя главными компьютерами (хостами), состоит из трех частей:

- 32-битового идентификатора сети (ID),
- 48-битового идентификатора хоста,
- 16-битового номера порта.

Терминология, применяемая для адреса XNS — сетевой идентификатор, идентификатор хоста и номер порта — аналогична терминологии адреса TCP/IP, который также состоит из сетевого номера, номера хоста и номера порта.

Компоненты адреса NetWare

Адрес NetWare, который имеет длину 12 байт и передается по IPX, включает в себя:

- 32-битовый сетевой номер, называемый *сетевым адресом* (*Network Address*) на утилите MONITOR NLM. Эта часть адреса используется для адресации индивидуальных рабочих групп, или сегментов LAN;
- 48-битовый узловой адрес, называемый так же на утилите MONITOR. Эта часть представляет собой уникальный (аппаратный) адрес для адресации узлов в сегменте LAN;
- 16-битовый номер гнезда, который называется номером порта в терминологии TCP. Эта часть используется для мультиплексирования функций в рамках одного сетевого узла.

ПРИМЕЧАНИЕ: Это понятно, так как с использованием IPX или SPX на сервере можно обслуживать несколько удаленных пользователей, или клиентов. Это требует наличия метода идентификации данных, связанных с каждым уникальным удаленным процессом. Для этой цели используются номера портов. Для идентификации номеров портов, которые были постоянно присвоены конкретным протоколам, используется термин *"хорошо известные порты"* (*well-known ports*) или *"хорошо известные адреса"* (*well-known addresses*).

Аналогией является глобальный порт RPC *Portmapper* протоколов TCP/IP, или хорошо известный адрес (номер порта) 111. Все данные, получаемые на этом серверном порте, являются данными, связанными с портом RPC Portmapper, и идут на специальную обработку, которую осуществляет этот резервированный порт. Клиент, который хочет послать дейтаграмму на демон, или процесс, порта Portmapper на этом сервере, определяет этот уникальный номер порта в дейтаграмме. Уникальный исходный адрес *сетевого номера/номера хоста* в дейтаграмме сообщает серверу, от какого клиента были посланы эти данные. Местный процесс клиента, или номер порта, также включается в диаграмму. Таким образом, и сервер может отвечать клиенту уникальным образом.

Обычно эти номера выражаются шестнадцатиричными числами. Одна шестнадцатиричная полубайтовая (*nibble*) цифра представляет 4 двоичных бита.

Сетевой номер (*network number*) NetWare определяет сетевой кабельный сегмент или локальную сеть (LAN), а узловой адрес определяет уникальный узел, или хост, внутри этого сегмента. *Номер гнезда* используется так же, как и номер порта с TCP и протоколом дейтаграммы пользователя (User Datagram Protocol — UDP).

На рис. 4-1 показан простой сетевой сегмент DataWare с двумя узлами и их уникальными адресами.

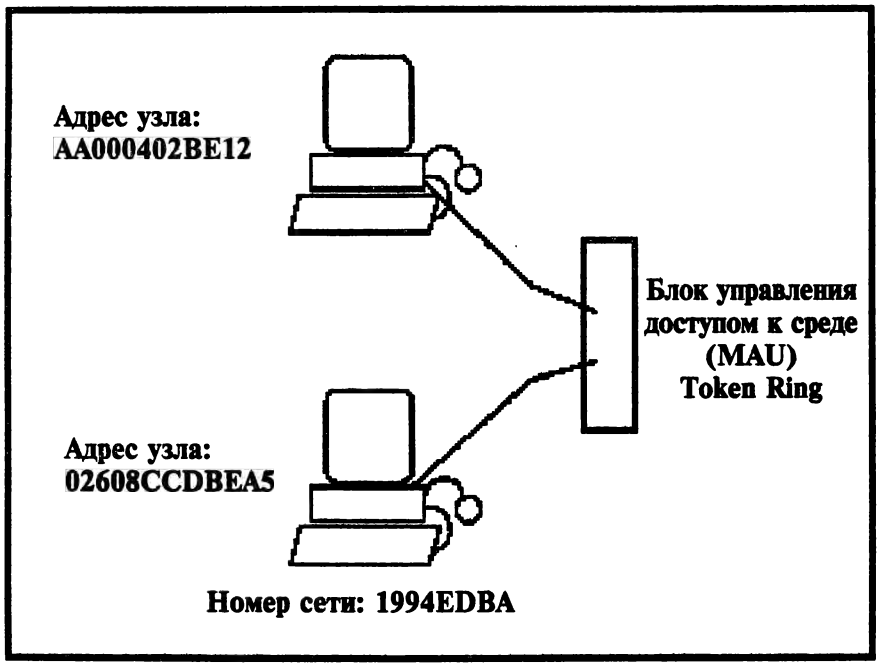


Рис. 1-4. Сетевой сегмент NetWare.

Внутренний сетевой номер и узловой номер NetWare

Внутренняя сеть внутри NetWare является отличительной характеристикой версии NetWare 3.0 и выше. Внутренняя сеть — это логическая сеть, которая отправляет по различным маршрутам пакеты в физическую сеть, с которой соединен файл-сервер.

Внутренний сетевой адрес, аналогичный "внешнему" адресу NetWare, включает в себя уникальный внутренний *сетевой номер*, а также узловой номер, который по умолчанию всегда равен 1. При инсталляции V3.1x NetWare вам будет задаваться требование обеспечить внутренний сетевой номер. Это — шестнадцатиричное число из 1-8 цифр.

Операционная система LAN Manager

Введение

Сетевая операционная система LAN Manager фирмы Microsoft реализована в нескольких "ипостасях": DOS LAN Manager, версия 3.1 Windows и Windows для рабочих групп (Windows for Workgroups) версия 3.11, усовершенствованный сервер новой технологии — New Technology Advanced Server — NTAS и Windows NT для клиентов и UNIX Open VMS для серверов. Мы сделаем краткий обзор архитектуры

LAN Manager, а затем рассмотрим лежащие в его основе средства передачи и протоколы.

Версия 1.0 операционной системы LAN Manager для DOS была выпущена в 1987 году примерно в то же самое время, что и версия MS-DOS V3.3 и версия V1.0 Microsoft OS/2. Когда эта книга сдавалась в печать, текущей версией для систем DOS была версия 2.2 LAN Manager для MS-DOS. Образование сетей с помощью NTAS и NT реализует более сложное применение операционной системы LAN Manager.

Компоненты клиента в LAN Manager

Клиент LAN Manager в DOS и Windows имеет в качестве основы *редиректор* (*redirector*) сети, который "делает ответвление" в местной системе для прослушивания системных вызовов, которые предназначены для сетевого дисковода. Если редиректор "слышит" такие вызовы, то он перехватывает вызовы и направляет данные в них на сервер для выполнения. Редиректор также распаковывает при необходимости все возвращенные результаты. Ключевой компонент кода редиректора для MS-DOS впервые был упакован в выпуске версии V3.1 DOS в 1984 году как набор системных вызовов DOS.

Программный интерфейс NetBIOS обеспечивает унифицирующий уровень кода для связи "клиент-сервер" в *виртуальной цепи* (*virtual-circuit* communications) независимо от лежащих в его основе транспортных средств. Виртуальная цепь — это сеанс или канал между двумя взаимодействующими компьютерными узлами. Таким образом, NetBIOS представляет собой уровень управления сеансами для LAN Manager.

LAN Manager включает в себя носители сетевых данных, или средства передачи данных (транспортные средства). Они включают в себя NetBEUI и TCP/IP, а также протокол DECnet в сетевой операционной системе PATHWORKS фирмы DEC, основанной на LAN Manager. Эти транспортные средства "сидят" под сеансовым уровнем NetBIOS по отношению к модели OSI.

Драйверы, которые в действительности посылают и принимают кадровые данные в сетевом "проводе", — это драйверы управления доступом к среде передачи (Media Access Control — MAC) и, как правило, они согласованы с NDIS. Кроме того, они также рассматриваются как часть сетевой операционной системы LAN Manager.

LAN Manager также включает в себя API, который изолирует программиста от нижележащих сетевых протоколов и транспортных средств и драйверов MAC. Эти API "разговаривают" с кодом редиректора и используют протокол блока сообщений сервера (Server Message Block — SMB) для осуществления ввода/вывода файлов и других рутинных операций в сети. В настоящей книге не приводится ни одного примера программы "клиент-сервер" LAN Manager.

LAN Manager также включает в себя другие *виды сервиса* (*services*), т.е. кодовые фрагменты, которые реализуют конкретный вид сервиса у клиента или сервера. Примером является сервис NETLOGON, который вводит пользователей в домен,

а также реализует повтор учета от первичных серверов к резервным и действующим серверам. LAN Manager также включает в себя утилиты и интерфейсы пользователей, которые помогают в реализации и управлении средой LAN Manager. В этой книге они не рассматриваются.

Домены операционной системы LAN Manager

Операционная система LAN Manager использует *домены* (*domains*) для организации сетей. Домен — это группа сетевых узлов, организованная по схожей функциональности. Например, домен под названием ENGINEERS может состоять из имен всех инженеров для конкретного программного продукта. Имена доменов — это действующие имена групп NetBIOS, которые используются всеми узлами, участвующими в домене. Это групповое имя может соответствовать групповому (вариант широковещательного) адресу в физическом уровне, такому, например, как групповой адрес сети Ethernet. Это позволяет узлам участвовать в рассылке пакетов ко всей группе.

Основная выгода от группирования доменов в операционной системе LAN Manager — это то, что серверы в домене образуют единую единицу управления. Каждый домен имеет одну задающую базу данных, содержащую учет работы пользователя и группы, а также параметры политики безопасности. Эта база данных, называемая Системой учета пользователей (User Accounts System — UAS), находится в первичном контроллере доменов (Primary Domain Controller — PDC) и периодически *дублируется* (*replicated*), или копируется, в каждый *резервный* (*backup*) и *действующий* (*member*) сервер в домене. Таким образом, администраторам необходимо управлять только одним учетом для каждого пользователя; все изменения в учете происходят на PDC и дублируются по всему домену на резервные и действующие серверы.

Домены операционной системы LAN Manager реализуют один из двух типов безопасности: *на уровне пользователя* (*user-level*) и *на уровне разделения* (*share-level*). При осуществлении безопасности на уровне разделения доступ в режим разделения операционной системы LAN Manager управляется с помощью пароля, установленного для совместного режима (режима разделения). Безопасность на уровне пользователя применяется наиболее часто, так как она обеспечивает повышенную степень безопасности через учет вхождений в систему механизма привилегий, а также элементы списка управления доступом (Access Control List — ACL), находящиеся в каталогах и/или файлах. Эти элементы называются ACE (Access Control Elements).

Эти списки управления доступом связывают список разрешений с пользователями или группой пользователей. Объединение пользователей в группы является важной частью процесса управления в домене операционной системы LAN Manager. Пользователь рабочей станции должен сначала войти в домен, для того чтобы осуществить правильный доступ к большинству видов сервиса в домене, где безопасность реализована на уровне пользователя. Сетевые администраторы обычно группируют пользователей в группы, затем распределяют элементы ACE группового уровня по каталогам уровня коллективного пользования (share-level directories) для осуществления управления доступом к этому разделяемому каталогу.

PATHWORKS для OpenVMS (OC LAN Manager)

Описание сетевой операционной системы PATHWORKS

Сетевая операционная система для сетей персональных компьютеров фирмы Digital Equipment Corporation носит имя PATHWORKS. Последние выпуски этой сетевой операционной системы — версия 5.0, 5.0A, 5.0B, 5.0C и соответствующие обновления — на OpenVMS называются сетевой операционной системой PATHWORKS для OpenVMS (LAN Manager) и, следовательно, используют в качестве своей основы архитектуру операционной системы LAN Manager. То же самое справедливо для операционной системы PATHWORKS для продукта OSF/1 (LAN Manager) фирмы DEC.

Фирма Open Software Foundation — OSF сделала OSF/1 своим первоначальным предложением операционной системы для компьютерной промышленности. Фирма OSF лицензирует OSF/1 поставщикам для ее переноса на их компьютерные платформы. Характеристики OSF/1 включают в себя симметричную обработку в мультирежиме (symmetric multiprocessing — SMP) и позволяют выполнять одновременно отдельные ветви одного приложения. OSF/1 включает в себя характеристики, которые первоначально имелись в Berkeley Software Distribution (BSD) и System 5 UNIX, а также в сетевой файловой системе (Network File System — NFS) фирмы Sun Microsystem.

Большая часть функций версии 2.2 операционной системы LAN Manager фирмы Microsoft применяется в продуктах версии 5.x фирмы DEC на OpenVMS, в то время как реализация OSF LAN Manager является полной реализацией LAN Manager V2.2.

Фирма DEC также применяет PATHWORKS на Windows NT и NTAS — версию 4.1 для NT версии 3.1 и версию 4.2 для Windows NT и NTAS версии 3.5.

Код сервера для PATHWORKS для OpenVMS и PATHWORKS для OSF/1 также работает на серверах Alpha фирмы DEC. ALPHA представляет собой 64-битовую микросхемную архитектуру компьютера с уменьшенным набором команд (Reduced Instruction Set Computer — RISC), разработанную фирмой DEC. Клиент версии 5.1 PATHWORKS (версия 5.1 была текущей версией клиента, когда эта книга была опубликована) поддерживает персональные компьютеры клиентов с DOS и Windows (V3.1), а также Windows for Workgroups (версия 3.11) и клиентов NetWare (версия 3.11 и ниже).

Версии 4.x PATHWORKS фирмы DEC для OpenVMS используют протоколы Microsoft Networks (MS-NET). MS-NET была предшественницей LAN Manager и использует основной поднабор протоколов блока сообщений сервера (Server Message Block — SMB) OC LAN Manager. Сервер OpenVMS V4.x PATHWORKS представляет собой сервер MS-NET. Таким образом, он не поддерживает характеристики LAN Manager, такие как домены и вход в систему (logon) LAN Manager.

Реализация V5.0C операционной системы LAN Manager на сервере OpenVMS, осуществленная фирмой DEC, предлагает большинство из стандартных характеристик версии 2.2 LAN Manager. Одним исключением является дублирование файла и директория на домене. Фирма DEC также обеспечивает поддержку LAN Manager на VAX-кластерах OpenVMS. Фирма DEC также улучшает модель безопасности LAN Manager, используя преимущества усложненных характеристик безопасности и доступа к файлу операционной системы OpenVMS для добавления другого уровня безопасности при доступе к файлу в LAN Manager. Этот уровень безопасности называется *Хост*-безопасностью.

ПРИМЕЧАНИЕ: Кластеры — это многокомпьютерные системы OpenVMS, которые связаны друг с другом специальным программным обеспечением для реализации общего доступа к дисковым внешним устройствам и единым пунктом управления учетом для действительных членов кластера.

ManageWORKS

Продукт PATHWORKS, версия 5, фирмы DEC является уникальным в том смысле, что большинство функций сетевой операционной системы может исполняться непосредственно от персонального компьютера клиента с помощью Windows. Версия 5.1 PATHWORKS включает в себя *ManageWORKS*, утилиту Windows у клиента, что позволяет ему управлять доменами LAN Manager, а также серверами NetWare (версия 5.1 и ниже с версией 5.1 PATHWORKS).

Программа просмотра (*browser*) ManageWORKS — это кодовый сегмент, который "видит" LAN Manager и другие виды сервиса сети и представляет эти виды сервиса в графической форме для облегчения взаимодействия. Она позволяет вам:

- просматривать многооперационные домены, виды сервиса и директории,
- управлять видами сервиса LAN Manager и NetWare,
- надлежащим образом представлять все сетевые ресурсы и манипулировать ими,
- определять ваши собственные номера для сетевых ресурсов.

ManageWORKS имеет также компоненту Workstation Manager, которая позволяет администратору сети управлять центральной базой данных с помощью шаблонов начальной загрузки рабочей станции. Сетевое устройство управления (менеджер) может изменять любой шаблон группы компьютеров или рабочих станций в любой момент, осуществляя автоматические обновления шаблонов начальной загрузки, после того как пользователь вновь загружает свою рабочую станцию с помощью этого шаблона.

ОС PATHWORKS V5 LAN Manager DOS, клиенты Windows

Клиенты ОС PATHWORKS версия 5.1 — в DOS или Windows V3.1 — могут быть конфигурированы для нескольких транспортных средств одновременно. Эти транспортные средства включают в себя DECnet, TCP/IP, NetBEUI и IPX/SPX.

Конфигурация клиентов PATHWORKS поддерживает одновременную загрузку любой комбинации вышеупомянутых транспортных средств. Она также поддерживает ряд архитектур канального уровня для применимых транспортных средств. Среди них NDIS, открытый интерфейс канала данных (Open Data Link Interface — ODI) фирмы Novell и управление каналами данных (Data Link Control — DLC) фирмы IBM.

Дальнейшее обсуждение функций и операций PATHWORK лежит за пределами этой книги, так как в этой книге внимание сосредоточено на программировании гнезд и именованных программных каналов. Мы рассматриваем сеансовый уровень программного обеспечения NetBIOS, затем переходим к TCP/IP, на что особо указывается в главах книги, посвященных программированию.

ПРОТОКОЛЫ, ИНТЕРФЕЙСЫ ПРИКЛАДНОГО ПРОГРАММИРОВАНИЯ И СРЕДСТВА ПЕРЕДАЧИ ДАННЫХ СЕТЕВЫХ ОПЕРАЦИОННЫХ СИСТЕМ

Введение

Упомянув о сетевых операционных системах, которые соответствуют тематике этой книги, а также о средствах межсетевого обмена пакетами, рассмотрим теперь другие протоколы и средства передачи данных (транспортные средства), которые осуществляют работу в среде этих сетевых операционных систем.

Сначала мы рассмотрим NetBIOS — интерфейс сеансового уровня для LAN Manager — затем подробно рассмотрим набор протоколов TCP/IP, так как он является транспортным средством, на котором мы несколько позже в этой книге продемонстрируем программы. Мы также кратко рассмотрим транспортные средства DECnet, хотя программа с DECnet не приводится.

NetBIOS

Введение

Сетевая базовая система ввода/вывода (Network Basic Input/Output System — NetBIOS) обеспечивает интерфейс прикладного программирования сеансового уровня, который относится к конкретному средству передачи (транспортному средству) сетевого стека. Стек может быть определен как совокупность сетевых прикладных программ, которые включают в себя сетевую систему доставки сетевой операционной системы.

NetBIOS была введена фирмой IBM в 1984 году в качестве части сети программной поддержки компьютерной локальной сети. Вышеприведенное определение подразумевает, что каждое транспортное средство должно применять свой собственный код, специфичный относительно NetBIOS — так это и есть на самом деле.

Определение/использование NetBIOS

Программное обеспечение NetBIOS обеспечивает сетевые функции ввода/вывода для интерфейсов прикладного программирования верхнего уровня (LAN Manager) в виде команд через интерфейс прикладного программирования (API). Команды NetBIOS CALL (для клиента) и LISTEN (для сервера) позволяют установить сеансы между двумя сообщающимися компьютерными узлами. LAN Manager использует NetBIOS в качестве драйверного устройства для обработки элементов нижнего уровня сетевой связи. На рис. 1-5 показан интерфейс NetBIOS с сетью в его отношении к модели OSI.

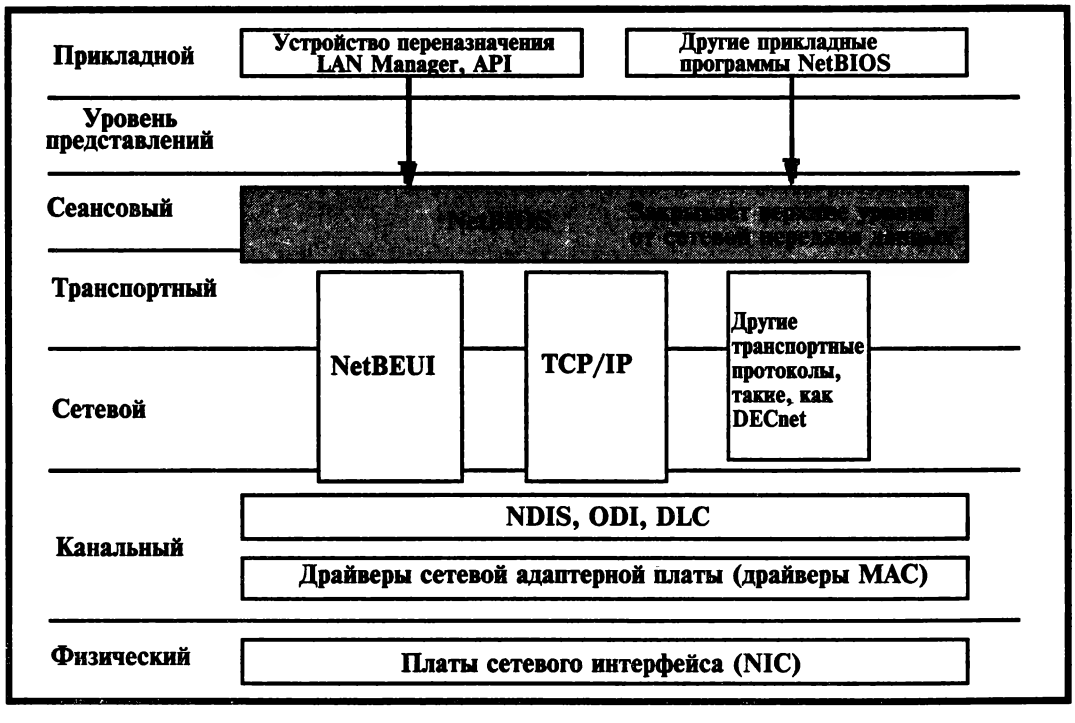


Рис. 1-5. Интерфейс NetBIOS в модели OSI.

Функции NetBIOS

Интерфейс NetBIOS имеет следующие три основные функции:

- регистрирует уникальные имена компьютера и неуникальные *групповые имена* в сети;
- устанавливает и поддерживает виртуальные цепи ("сеансы" или каналы) между двумя сетевыми узлами;
- поддерживает дейтаграммную связь как в широковещательном (трансляционном), так и в групповом смысле.

Регистрация имен

NetBIOS предлагает программирование вызовов для объявления (регистрации) имен в сети. Рабочая станция может зарегистрировать уникальный компьютерный номер, которого теоретически нет ни у одного другого узла в сети. Хотя дубликаты компьютерных имен NetBIOS могут иметь место в локальной компьютерной сети при определенных обстоятельствах, чаще всего при нормальных обстоятельствах этого не происходит. Сервер LAN Manager может регистрировать групповое имя, которое, как упоминалось ранее, позволяет всем серверным узлам в одной и той же группе получать дейтаграммы, которые были переданы в групповом (multicast) режиме по адресу нижнего уровня, или физического уровня, связанному с этим групповым именем.

Компьютерные номера NetBIOS обеспечивают программиста средством, дающим ему возможность не обязательно связываться непосредственно с конкретным аппаратным адресом Ethernet или эстафетного кольца и т.д. Это означает, что программа может передавать определенные данные от одной станции к другой с помощью компьютерных имен вне зависимости от того, с каким физическим адресом — или с каким NIC — связаны эти имена.

Виртуальные цепи

Виртуальная цепь — это надежный полнодуплексный канал "от пункта к пункту" или "от узла к узлу". Слово "канал" (link) употребляется здесь для тех, кто привык к среде DECnet. Термин "надежный" означает, что передача данных осуществляется с подтверждением приема, а термин "полнодуплексный" означает, что связь является двусторонней.

Когда устанавливается виртуальная цепь, через редиректорный код LAN Manager на этой виртуальной цепи может быть установлено одно или несколько *соединений* (*connection*). Соединение представляет собой установление переназначения отображения буквы драйва DOS в каталог сервера, называемый *долей* (*share*) каталога по терминологии LAN Manager.

Поддержка дейтаграммной связи

Дейтаграмма представляет собой передачу пакета без установления логического соединения (connectionless) или такого, который передается независимо от всех других пакетов сети. Дейтаграммная передача является ненадежной в том смысле, что она не требует подтверждения приема. Одной из форм дейтаграммной передачи является групповая (multicast).

NetBIOS поддерживает дейтаграммную передачу, которая позволяет прикладной программе передавать неподтвержденные данные на одно, несколько (*в групповом режиме*) или на все (*в широковещательном режиме*) имена в сети без гарантии доставки сообщения.

Передачи в групповом режиме NetBIOS в случае работы с Ethernet привязываются к групповым адресам на канальном уровне Ethernet. Например, транспортное средство Фазы IV DECNet фирмы DEC использует тип протокола из 80-40 шестнадцатиричных чисел при передаче дейтаграммы группового режима NetBIOS в кадре Ethernet II. Так как фирма DEC присвоила групповой адрес 09-00-2B-00-00-07 коду эмулятора PATHWORKS NetBIOS на Ethernet, это действительный адрес, который будет помещен в заголовок управления доступом к среде передачи данных (MAC) кадра Ethernet в качестве адреса получателя, когда функция групповой дейтаграммы NetBIOS вызывается кодом LAN Manager. Все узлы, способные принимать дейтаграммы по этому адресу, будут принимать эти сообщения.

Клиентский интерфейс TCP/IP NetBIOS фирмы DEC

Пример архитектуры NetBIOS в том виде, в котором она реализована в версии 5.1 у клиента TCP/IP DOS и Windows, показан на рис. 1-6:

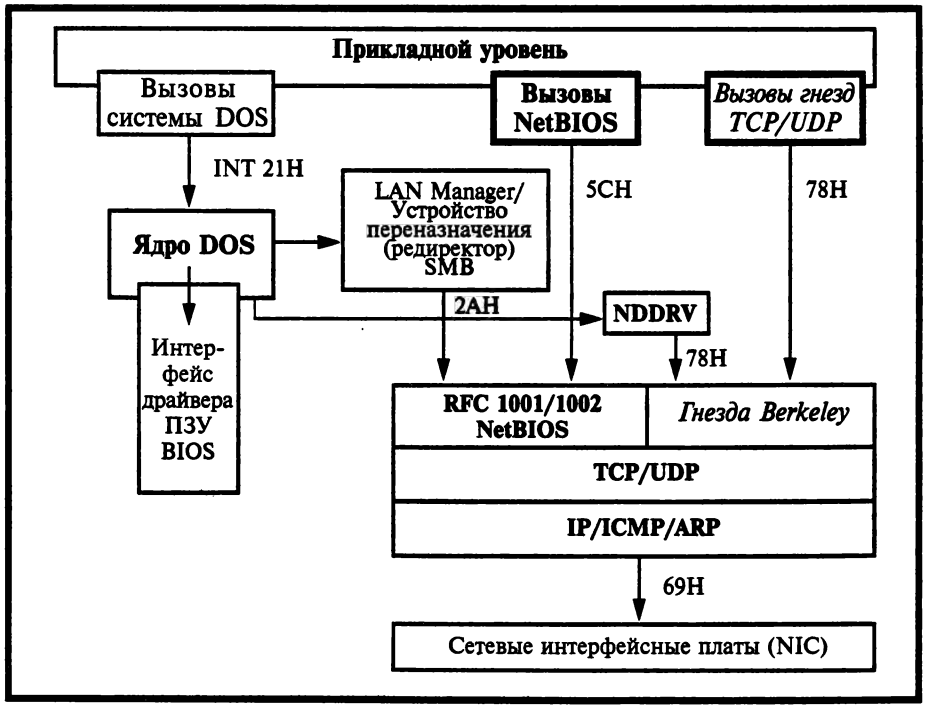


Рис. 1-6. Архитектура NetBIOS у клиента TCP/IP DOS и Windows фирмы.

Отметим, что, когда вы пишете программы TCP/IP от гнезда к гнезду, уровень кода NetBIOS обходится и не используется. Гнездовой интерфейс разбивается на уровни как раз над транспортным средством, на котором он реализуется. Гнездовое программирование TCP является ядром, на котором сконцентрировано основное внимание этой книги.

ПРИМЕЧАНИЕ: Запрос на комментарий (Request for Commentary — RFC) является общим способом UNIX-запроса на ввод для протоколов стандартизации и интерфейсов прикладного программирования. Определения RFC 1001 и 1002 определяют NetBIOS в том виде, в котором она реализована на TCP/IP в среде Internet. Кроме того, числа на диаграмме являются шестнадцатиричными числами прерывания, используемыми командой INT ассемблера Intel, для того чтобы осуществить программное прерывание для обращения к программному (например, NetBIOS) API.

Выводы относительно NetBIOS

NetBIOS — это интерфейс сеансового уровня под интерфейсом прикладного программирования (API) LAN Manager, а также интерфейса прикладного программирования сетевой операционной системы PATHWORKS фирмы DEC. Таким образом, все переназначения сетевого драйва на сервер LAN Manager происходят над установкой сеанса NetBIOS.

- NetBIOS предусматривает групповой режим — одну из форм широковещательного режима, при которой сообщения передаются на специальную группу главной системы — в сетевой среде с имеющимися в сети командами.
- NetBIOS изолирует прикладные программы LAN Manager от необходимости иметь дело с нижними уровнями сетевого ввода/вывода.
- Каждое транспортное средство под NetBIOS должно реализовывать некоторый код, зависящий от транспортного средства, для взаимодействия с NetBIOS.
- Регистрация имени NetBIOS в сети — как уникальных компьютерных номеров, так и неуникальных групповых номеров — является важной частью NetBIOS и сетевых операционных систем, которые зависят от интерфейса прикладного программирования (API) NetBIOS.

Протоколы TCP/IP

Введение

Теперь рассмотрим краткую историю набора протоколов: протоколов управления передачей/межсетевых протоколов (TCP/IP) и рассмотрим работу некоторых внутренних элементов набора.

TCP/IP предлагается с любым продуктом LAN Manager фирмы Microsoft. Фирма Digital Equipment Corporation включает свой транспортный стек клиента TCP/IP в версию 5.1 PATHWORKS для продуктов DOS и Windows.

Фирма DEC предлагает поддержку TCP/IP на своих серверах OpenVMS PATHWORKS через реализацию третьей стороны или же через собственный продукт TCP/IP. Клиенты TCP/IP PATHWORKS могут соединяться через LAN Manager и/или блоки сообщений сервера (SMB) основного устройства переназначения

на PATHWORKS для серверов OSF/1, PATHWORKS для серверов OS/2, а также PATHWORKS для SCO UNIX и PATHWORKS для серверов ULTRIX. Фирма DEC также предлагает поддержку именного доступа к серверам, а также доступ к серверам PATHWORKS в глобальной сети (Wide Area Network — WAN) с помощью утилиты INETNAME у своих клиентов TCP/IP.

Межсетевые протоколы DARPA

Первым набором сетевых протоколов, реализованных с версией 4.2 UNIX фирмы Berkeley Software Distributions (BSD), был набор межсетевых протоколов DARPA. В версии V4.3 BSD были реализованы протоколы Xerox Network System. Текущей версией Berkeley UNIX к моменту написания этой книги была версия V4.4.

Межсетевые протоколы DARPA были разработаны при спонсорстве Агентства передовых научно-исследовательских проектов министерства обороны США (Defense Advance Research Projects Agency — DARPA). Объединенная сеть DARPA была адаптирована к Ethernet, чисто дейтаграммной сети, которая не обеспечивает гарантии доставки дейтаграмм. В чисто диаграммной сети кадры Ethernet идут к каждому узлу Ethernet, однако они обрабатываются только по соответствующему адресу получателя.

Термин *Интернет* (*Internet*) — "межсетевой" — относится к совокупности сетей с пакетной коммутацией, связанных друг с другом с помощью сетевых шлюзов и протоколов, которые позволяют им логически функционировать в качестве единой большой виртуальной сети. Этот термин произошел от проекта DARPA Internet. Впоследствии политики изобрели выражение *Информационная магистраль* (*Information Superhighway*), которое относится к Internet из-за ее связности.

Примечание: "Пакетная коммутация" означает, что организованные "куски" данных, называемые *пакетами*, могут индивидуально отсылаться на любой узел получателя в сетевой среде передачи данных с помощью схемы адресации, встроенной в конструкцию пакетов. Системы под названием "маршрутизаторы" (routers) могут направлять (коммутировать) пакеты по соответствующим адресам в более крупных сетях.

Модель DARPA приводит к использованию по крайней мере трех протокольных уровней. Один работает в режиме "*от пункта к пункту*" между двумя хостами, ведущими обмен данных в сети. Он основан на протоколе нижнего уровня, который работает на основе принципа "скачок за скачком" (hop-by-hop), направляя каждое сообщение через промежуточные шлюзы по адресу получателя. В общем случае, существует по крайней мере один протокольный уровень над двумя другими — прикладной уровень. Эта трехуровневая система была названа эталонной моделью Арпанет (ARPANET Reference Model). Сеть ARPA была первоначальной сетью типа Internet DARPA, состоящей из каналов передачи данных со скоростью передачи 56 кбит/с.

Эталонная модель ARPANET

На рис. 1-7 представлена эталонная модель ARPANET.

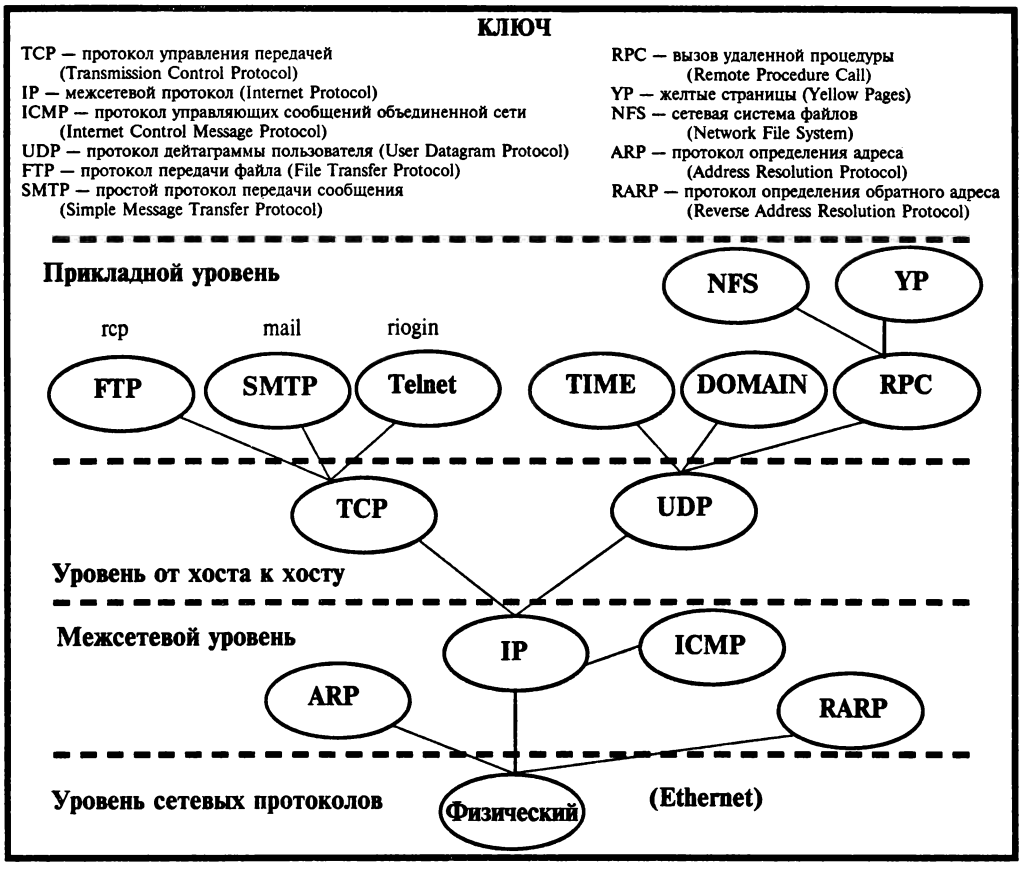


Рис. 1-7. Эталонная модель ARPANET.

Если бы эта модель была встроена в эталонную модель OSI, то вышеупомянутый межсетевой уровень отобразился бы в сетевой уровень OSI, в то время как уровень "хост-к-хосту" отобразился бы в транспортный уровень OSI.

Характеристики межсетевого протокола

Ниже приводятся некоторые характеристики доставки межсетевого протокола (IP):

- Уровень IP эталонной модели ARPANET отвечает за доставку дейтаграмм.

  Дейтаграммы — это пакеты, передаваемые в сети независимо от всех других. Дейтаграмма содержит адреса источника (отправителя) и получателя — адрес получателя может быть групповым (multicast) или широковещательным (broadcast), — а также данные. Они посылаются от одного главного компьютера (хоста) к другому, возможно, через взаимосвязывающие шлюзы. Шлюзы

часто нестрого определяются как системы, которые работают в одном или нескольких уровнях OSI. Шлюзы иногда взаимосвязывают две или несколько подсетей, которые используют различные транспортные протоколы. *Туннелирование* (*tunneling*) представляет собой термин, означающий, что один протокол туннелируется, т.е. помещается в секцию данных пакетных данных другого протокола. Шлюзы иногда используют этот способ для взаимосвязи локальных компьютерных сетей. Примером может служить шлюз "DECnet в Internet".

Примечание: PATHWORKS V5.1 использует термин *шлюз* (*gateway*) для обозначения "маршрутизатора" ("router"), когда он запрашивает информацию клиента TCP/IP. PATHWORKS в этом диалоге запрашивает шлюз по умолчанию (Default Gateway).

- Сервис IP является ненадежной, без установления логического соединения, системой доставки пакетов с наименьшими усилиями ("best-effort packet delivery system").

  Межсетевой сервис ненадежен, так как доставка сообщения не гарантируется. Не существует механизма, встроенного в уровень протокола IP, для подтверждения успешной доставки информации. Пакеты могут теряться, дублироваться или доставляться внутри этого уровня неупорядоченно. Кроме того, контрольная сумма — число, используемое для вычисления целостности пакета — обеспечивает обнаружение ошибки для заголовочной части ("хедера") дейтаграммы (IP), но не для сегмента данных дейтаграммы. Успешная доставка возлагается на протокольные уровни выше IP, такие, как TCP и UDP.

Виды сервиса сетевого уровня IP

Уровень IP отображается в сетевой уровень OSI. Как таковой, протокол IP обеспечивает следующие функции:

- адресацию к хосту (главному компьютеру),
- организацию маршрутов,
- при необходимости фрагментирование и повторную сборку пакетов,
- поддержку более высоких протоколов направлением на конкретный идентификационный слот протокола более высокого уровня, например номера порта.

Адреса IP

Межсетевой адрес хоста (*Internet host address*) представляет собой 32-битовое число, которое идентифицирует как сеть, в которой расположен хост (главный компьютер), так и хост или "узел" в этой сети. *Сетевой номер* (*network number*) аналогичен зоне (area) в узловом адресе DECnet, в то время как адрес хоста соответствует узловому адресу DECnet. Сетевые идентификаторы назначаются центральным агентством, а номера хоста обычно назначаются администратором каждой сети.

Это разделение сетевого номера и номера хоста позволяет шлюзам легко определять сетевой номер для организации маршрутов.

Таким образом, 32-битовый межсетевой адрес включает в себя сетевой номер и номер хоста, называемый иногда *адресом хоста* (*host address*). Старшие биты сетевой части адреса кодируются таким образом, что размер этой части адреса может меняться. Эти же самые старшие биты определяют *класс* адреса. Определены три первичных адресных класса — A, B и C — с битами старшего порядка 0, 10 и 110, соответственно. Четвертый класс, класс D, резервируется для групповой адресации, в то время как пятый класс, класс E, резервируется для использования в будущем.

Классы A, B и C содержат, соответственно, по 8, 16 и 24 бита, используемых для сетевой части адреса Internet. Каждый класс использует меньшее число битов, необходимых для "хостовой части" (части, относящейся к хосту), чем более старшие классы, и поэтому поддерживает меньшее количество хостов.

На рис. 1-8 приведены три первичных класса сетевых адресов IP:

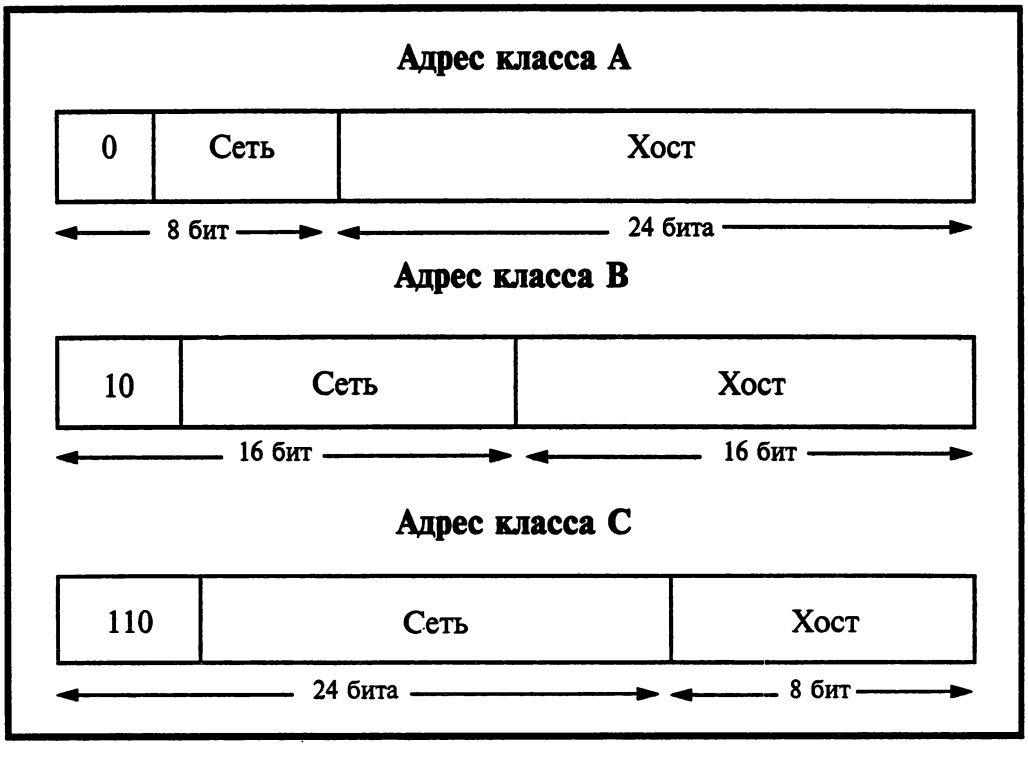


Рис. 1-8. Первичные классы сетевых адресов IP.

Обычно численное представление межсетевого адреса имеет следующий вид: каждое 8-битовое поле адреса обозначается десятичным числом, отделенным от других полей точкой. Например, адрес объединенной сети (Internet) 128.45.71.33, класса B, представлен на рис.1-9.

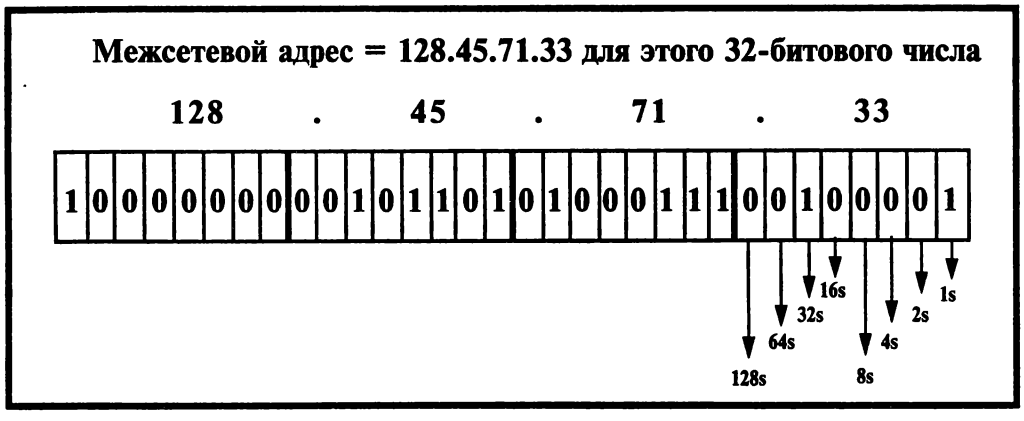


Рис. 1-9. Диаграмма межсетевого адреса.

Межсетевая схема адресации включает в себя *широковещательный* (*broadcast*) адрес, который является эталонным для всех хостов сети. Хотя в выпуске V4.2 BSD для широковещательных дейтаграмм использовалась хостовая часть, состоящая из всех нулей, после выпуска V4.2 BSD широковещательный межсетевой адрес был определен как хостовая часть, состоящая из всех единиц.

Версия 4.3 и более поздние версии BSD устанавливают широковещательный адрес для каждого интерфейса как хостовое значение, состоящее из всех единиц, однако она позволяет иметь и альтернативный адрес — все нули для хостовой части — для осуществления ретроспективной совместимости. Это также справедливо и для клиентов протоколов TCP/IP PATHWORKS, NetWare и LAN Manager.

Адресная схема IP также поддерживает *групповую* (*multicast*) адресацию. Групповая адресация представляет собой селективную форму широковещательного режима, в которой сообщения передаются группе хостов, а не всем хостам сети. Члены группы могут быть распределены по отдельным физическим локальным сетям; таким образом, групповая дейтаграмма может быть утеряна или передана вне установленного порядка, как и любая дейтаграмма IP.

Членство в группе группового режима IP является динамическим, т.е. хост может присоединяться к группе или выходить из нее в любое время. Каждая группа группового режима имеет уникальный адрес *класса* D, в котором первые четыре старших бита равны 1110, а остальные 28 битов могут быть любым числом, представляющим группу группового режима. Некоторые групповые адреса, называемые *хорошо известными* (*well-known*), присваиваются органами управления объединенной сетью, в то время как другие доступны для временного использования. Групповой режим использования межсетевого протокола может применяться в одной физической локальной компьютерной сети или же по всей объединенной сети с помощью шлюзов, которые опознают адресацию в групповом режиме использования межсетевых протоколов.

Протокол управляющих сообщений объединенной сети (Internet)

Протокол управляющих сообщений объединенной сети (Internet Control Message Protocol — ICMP) позволяет шлюзам и хостам обмениваться информацией начальной загрузки и сообщать об ошибках, найденных в процессе обработки дейтаграмм. Хотя ICMP использует основные транспортные средства межсетевых протоколов, как если бы это были протоколы высокого уровня, ICMP является интегральной частью межсетевого протокола и должен реализовываться каждым модулем IP. Протокольный уровень IP сам имеет встроенный код для генерирования дейтаграмм ICMP, когда это необходимо. Шлюзы посылают сообщения ICMP, когда они не могут доставить дейтаграмму или для того чтобы дать указание хосту использовать другой шлюз. Обычно хосты посылают сообщения ICMP для проверки "целостности" сети.

Некоторые из применений ICMP таковы:

- Эхо-запрос/эхо-ответ (пробный шар — *ping*). — Одной полезной диагностической утилитой является программа UNIX Packet Internet Grouper, которая посылает сообщения *эхо-запроса* (*Echo request*) ICMP на специальный механизм. По приему эхо-запроса получатель возвращает *эхо-ответ* (*echo reply*) ICMP. Использование "пробного шара" является отличным способом проверки целостности сетевых линий от одного хоста к другому, а также для того чтобы увидеть, в активном ли состоянии находится тот или иной конкретный хост и правильно ли он работает, если известно, что целостность линий сети в порядке, т.е. вы общаетесь с другими хостами по одному и тому же сетевому сегменту. "Пробный шар" проверяет как шлюзы, так и системы хостов.
- Получатель недоступен. — Если шлюз не может доставить дейтаграмму из-за того, что получатель неактивен или нет маршрута, он возвращает отправителю сообщение *получатель недоступен* (*destination unreachable*). Сообщение об ошибках ICMP включает в себя межсетевой заголовок дейтаграммы, который включает состояние ошибки. По приему сообщения об ошибке ICMP хост проверяет возвращенный заголовок для определения того, какой получатель вызвал затруднения.
- Обрыв источника (Source Quench). — Если шлюз должен выбросить дейтаграмму из-за отсутствия ресурсов, он посылает на источник дейтаграммы сообщение *обрыв источника* (*source quench*). Это сообщение посылается, когда пакеты просто выбрасываются приемником.
- Переадресация. — *Переадресация* (*redirect*) посылается хостом на шлюз, когда пакет IP мог быть послан на более приемлемый шлюз. Однако пакет тем не менее направляется далее шлюзом, который послал сообщение о переадресации. Само сообщение о переадресации содержит адрес более приемлемого шлюза.

Протокол определения адреса (ARP)

Протокол определения адреса (Address Resolution Protocol — ARP) используется для нахождения адреса Ethernet хоста IP в той же самой сети. При использовании ARP специальный пакет запроса широковещательно передается хостом, который желает знать адрес Ethernet другого хоста IP в той же самой физической сети. Все хосты принимают этот запрос, но отвечает только определенный хост, заполняя адрес Ethernet приемника. Хост — автор широковещательного запроса — принимает ответ и обрабатывает его соответствующим образом. ARP не является протоколом IP. Пакеты ARP не имеют заголовков IP. Они передаются в сегменте данных (data portion) кадра Ethernet.

Дейтаграммный протокол пользователя (UDP)

Дейтаграммный протокол пользователя (User Datagram Protocol — UDP) — это простой транспортный протокол с уровнем выше IP.

UDP обеспечивает сервис, ориентированный на дейтаграммы. Он обеспечивает прикладные программы минимальными средствами, необходимыми для доступа к исходному сервису доставки IP. UDP не обеспечивает упорядочения, контроля ошибок и контроля потока, как это делает TCP.

Протокол UDP позволяет пересылать дейтаграммы между процессами, выполняющимися на различных хостах IP. UDP добавляет возможность осуществления различения между групповыми получателями на машине с помощью использования *портов* (*ports*), и он осуществляет вычисление контрольных сумм для обеспечения целостности данных.

Аналогично дейтаграммному протоколу IP протокол UDP ненадежен и не обеспечивает логического соединения. Доставка не гарантируется, т.е. *ненадежна* (*unreliable*). Каждая дейтаграмма обрабатывается независимо от всех других, т.е. *не имеет логического соединения* (*connectionless*).

Протокол UDP использует *порты* отправителя и *порты* получателя — пункты соединения с программным обеспечением у сервера и клиента — для мультиплексирования процессов. Межсетевая (Internet) прикладная программа должна уметь определять, с КЕМ она хочет установить связь. С точки зрения TCP программное обеспечение должно уметь *демультиплексировать* (*demultiplex*) поступающие сегменты данных на их соответствующие пункты назначения.

Наличие адресов источника и получателя является недостаточным, так как необходимо разрешать, например, групповые соединения TCP между одними и теми же двумя хостами. Как и TCP, UDP использует порты для решения проблем адресации. Каждый конец соединения опознается по своему адресу IP и 16-битовому *номеру порта*. Порты называют виды сервиса и клиентов, а хост инициирует соединение — обычно осуществляет соединение с серверами в *хорошо известных*

портах. Эти порты фиксированы в стандартизованных приложениях UNIX и редко меняются, что делает возможным их безопасное использование в программах в качестве постоянных.

Хорошо известные порты UDP

Порты 0-255 зарезервированы и приписаны к этим "стандартным" видам сервиса, или видам сервиса, которые являются стандартными для инсталляции UNIX, совместимой с BSD. Способность принимать дейтаграммы на резервированных портах ограничивается *корневой* (*root*) привилегией суперпользователя. Хорошо известные порты рассматриваются более подробно в разделе этой книги, посвященном программированию и рассматривающем гнезда TCP.

Некоторые общие порты UDP и связанные с ними протоколы представлены на рис. 1-10.

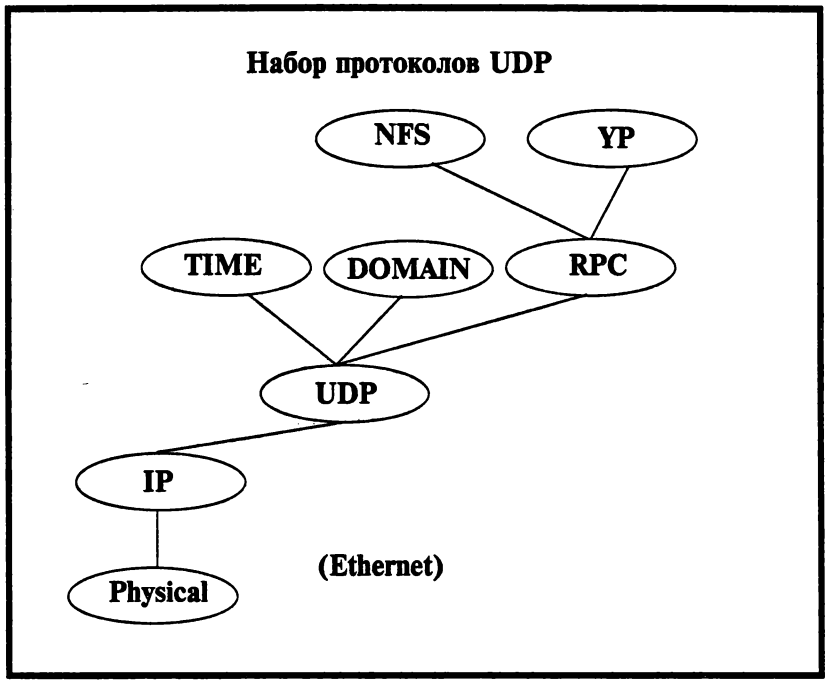


Рис. 1-10. Общие протоколы UDP.

Ниже приводятся некоторые общие протоколы UDP, связанные с ними порты и краткое описание каждой из зон:

- Эхо (порт 7). — Отсылка дейтаграммы; получение ее обратно.
- Время (порт 37). — Отсылка дейтаграммы; получение обратно времени в сетевом формате.

  Сетевое время: 32 бита.

  Количество секунд с 1 января 1900 года в универсальном времени.

- Система имени домена (Domain Name System — DNS) (порт 53). — Сервис каталога — адрес от имени хоста к IP.
- Тривиальный протокол передачи файлов (Trivial File Transfer Protocol — TFTP) (порт 69). — Стандартный протокол TCP/IP для передачи файла по UDP.
- BOOTP (порты 67, 68). — Дистанционный сервис первоначальной загрузки.
- Устройство отображения портов (Portmapper) при вызове удаленной процедуры (Remote Procedure Call — RPC) (порт 111). — Используется для реализации сервиса каталога, относящегося к Желтым страницам, и сетевой файловой системы (Network File System — NFS). Этот порт связан с функциями RPC, в которых программа может регистрировать свой программный номер, версию и номер порта UDP в рабочем процессе (daemon) устройства отображения портов (portmapper).

Как работает UDP

UDP предназначен для прикладных программ, в которых необходимо сохранить границы записей. Он вкладывается в сетевую модель практически так же, как TCP. Имеется заголовок UDP, и программное обеспечение сети ставит заголовок UDP перед данными точно так же, как заголовок TCP ставится перед своими данными. В случае исходящих пакетов UDP затем посылает данные на IP. Уровень IP добавляет заголовок IP, ставя номер протокола UDP в поле типа протокола в кадре Ethernet II или ставя соответствующий номер пункта доступа к сервису (Service Access Point — SAP) в однобайтовое поле SAP кадров IEEE 802.2.

Однако UDP не способен выполнять столько работы, сколько выполняет TCP. Он не разбивает данные в групповые пакеты. Он не отслеживает, что было им послано, для того чтобы осуществить в случае необходимости повторную отсылку сообщения. UDP предоставляет номера портов, так что некоторые программы могут сразу использовать UDP.

Протокол управления передачей (TCP)

Протокол управления передачей (Transmission Control Protocol — TCP) — это протокол, который преобразует исходный вид сервиса IP, ориентированный на дейтаграмму, — в котором дейтаграммы могут теряться, дублироваться или переупорядочиваться — в полнодуплексный, двусторонний надежный поток символов.

Следующие характеристики и их описания являются основой протокола TCP связки TCP/IP:

- Обеспечение надежной передачи *потока* (*stream*) символов.

  На прикладном уровне ставится задача передачи большого объема данных от одного хоста к другому. Одной целью межсетевых (Internet) протоколов является получение мощного и удобного для использования коммуникационного средства, которое позволяет программисту писать такие прикладные

программы. Одним из таких видов сервиса является *сервис надежного потока* (*reliable stream service*), который принимает поток 8-битовых групп, или байтов, и представляет его в удаленной прикладной программе точно в том же порядке, который был выработан отправителем.

TCP обеспечивает прикладные программы безошибочным полнодуплексным каналом связи. TCP обрабатывает все детали восстановления утерянных пакетов и обеспечивает целостность данных. Прикладные программы используют открытые связи TCP для отправления и приема данных на удаленные прикладные программы, а затем закрывают связь после ее окончания.

Обычно одна прикладная программа действует как сервер, ожидая открытия связи с ней удаленных прикладных программ, в то время как другая представляет собой активного клиента, который действительно инициирует связь с этим сервером. Это является классическим взаимоотношением между клиентом и сервером.

- Подтверждение приема с повторной передачей.

  Для того чтобы устранить ненадежность сервиса доставки, IP TCP использует алгоритм *квитирования с повторной передачей* (*acknowledgement-with-retransmission*). TCP делит входной поток на *сегменты* такого небольшого размера, чтобы они могли быть вложены в дейтаграммы IP. Каждый сегмент содержит последовательный номер, который идентифицирует данные, содержащиеся в сегменте. При получении сегмента TCP возвращает отправителю *подтверждение приема*, которое определяет, какой сегмент был получен. Если отправитель не получает подтверждения приема, то после некоторого ожидания он принимает решение о том, что данные были утеряны, и осуществляет повторную передачу.

- Контрольные суммы для обнаружения ошибки.

  Заголовок TCP содержит 16-битовую контрольную сумму, используемую для контроля целостности содержимого заголовка сегмента и данных. Эта контрольная сумма является необычной, так как она включает в себя поля заголовка IP. В качестве этапа вычисления контрольной суммы TCP добавляет *псевдозаголовок* (*pseudoheader*) к сегменту, который содержит адреса отправителя и получателя. Использование этого псевдозаголовка обеспечивает доставление сегмента к нужному получателю.

- Номера портов отправителя и получателя для мультиплексирования.

  Процесс *мультиплексирования* использует адрес и номер порта отправителя IP и адрес и номер порта получателя IP для установления связи. Все это определяет уникальные соединения TCP.

- *Скользящие окна* (*sliding windows*) для увеличения производительности.

  TCP, как и NetBEUI, использует протокол со скользящими окнами, в котором окно определяет данные, которые передал отправитель, но в отношении

которых получатель еще не подтвердил прием. Когда отправитель получает подтверждение приема для данных, в отношении которых ранее не было подтверждения приема, окно *продвигается вперед* (*advance*), представляя новые данные для передачи.

На рис. 1-11 показан протокол со скользящими окнами. В этом примере окну передачи разрешается представлять не более трех пакетов перед получением подтверждения принимающей станции о приеме сообщения. Таким образом, размер окна равен трем.

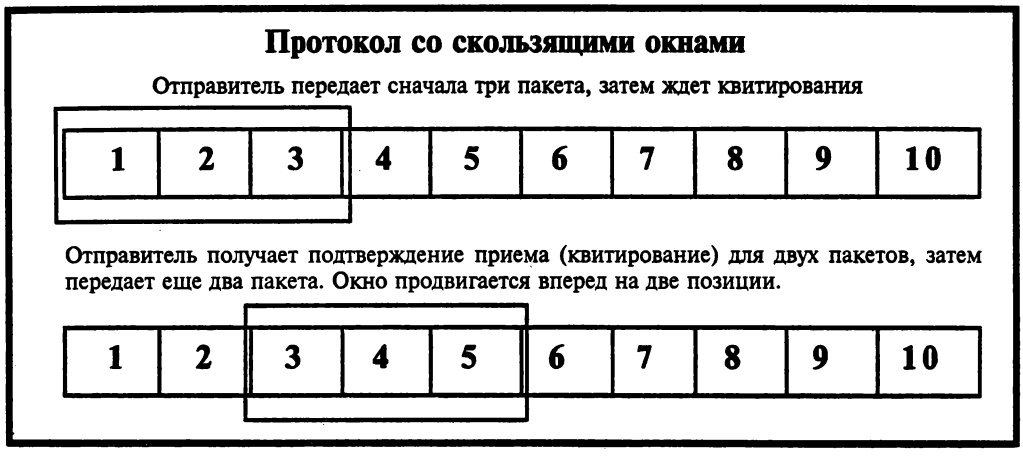


Рис. 1-11. Протокол со скользящими окнами.

DECnet

DECnet и DNA

DECnet является семейством связных программно-аппаратных комплексов фирмы Digital Equipment Corporation, которое позволяет компьютерным системам связываться в локальной компьютерной сети (LAN) или в широкомасштабной (глобальной) компьютерной сети (WAN). Фаза IV DECnet управляется архитектурной моделью фирмы DEC (Digital Network Architecture — DNA), которая является собственной связной моделью, разработанной фирмой DEC, и подобна модели OSI. Фаза V DECnet, последняя выпущенная версия DECnet, следует за моделью OSI для открытых систем и дублирует все уровни OSI.

DECnet реализует связную архитектуру по одноуровневому типу ("peer-to-peer"). Пользователи могут непосредственно связываться и делить ресурсы с любыми другими пользователями в сети без необходимости консультироваться с центральной системой управления. Однако DECnet используется во взаимосвязи "клиент-сервер" с PATHWORKS через LAN Manager, и уровни API NetBIOS выше DECnet. Отметим, что при этом описании одноуровневого принципа TCP/IP также является и архи-

тектурой одноуровневого типа, так как системы UNIX могут связываться друг с другом и делить ресурсы без необходимости консультирования с центральной системой.

Фаза IV DNA предлагает иерархическую модель, которая поддерживает связь между смежными уровнями с помощью точно определенных протоколов и интерфейсов. Модель Фазы IV DNA показана на рис. 1-12 и сравнивается с моделью OSI:

Модель DNA Фаза 4		Модель OSI	
6	Пользователь	7	Прикладной
5	Сетевые прикладные программы	6	Уровень представлений
4	Сетевой сервис	5	Сеансовый
3	Транспортные средства (сеть)	4	Транспортный
		3	Сетевой
2	Канал данных	2	Канальный
1	Физический	1	Физический

Рис. 1-12. Сравнение модели Фазы IV DNA с моделью OSI.

Физический уровень и уровень канала данных Фазы IV DNA напоминают по своей функциональности физический уровень и уровень канала данных OSI. Одно небольшое различие в физическом уровне может быть иллюстрировано таким примером:

DNA определяет Ethernet II как тип кадра по умолчанию для своего механизма доставки с определением множественного доступа с контролем носителя/обнаружением конфликтов (Carrier Sense Multiple Access/Collision Detect — CSMA/CD). OSI определяет типы кадров IEEE 802.3, 802.2 и 802.2 SNAP как средства доставки CSMA/CD.

Транспортный уровень Фазы IV DNA также включает в себя маршрутные определения для DECnet, а OSI четко различает функциональность сетевого и транспортного уровней. Сетевой сервисный уровень Фазы IV DECnet обеспечивает связной механизм, не зависящий от местоположения для обоих уровней выше этого сервисного уровня. Этот уровень отвечает за установление логических связей,

или сеансов, с помощью сетевого сервисного протокола (Network Services Protocol — NSP).

Фаза V возникла для поддержки как собственных протоколов DEC для предыдущих итераций DNA, так и многих протоколов, основанных на стандартах. Версия 5.1 PATHWORKS поддерживает только фазу IV DECnet на стороне клиента; PATHWORKS для сервера OpenVMS поддерживает Фазу V DECnet.

Связь от задачи к задаче

Вследствие того, что все компоненты DECnet реализуют протоколы DNA, вы можете иметь доступ к другим хостам сети DECnet независимо от типа операционной системы. Эти программные компоненты принимают вызовы, характерные для операционной системы, на местном узле и транслируют их в запросы для видов сервиса, характерных для сети. Затем компоненты DECnet на удаленном узле могут транслировать эти протокольные запросы в тот же самый набор сообщений.

Организация этой связи от компоненты к компоненте, или *связи от задачи к задаче* (*task-to-task communication*), является методологией для всех утилит DECnet. Эти утилиты включают в себя функции высокого уровня, такие как:

- Дистанционный доступ к файлу. — Эта функция включает в себя утилиту передачи файла, а также модуль дистанционной загрузки системы. Протокол, используемый для передачи файла — протокол доступа к данным (Data Access Protocol).

- Сетевая терминальная связь. — С компьютерного узла DECnet-DOS вы можете связаться с удаленным узлом OpenVMS работающей сети DECnet и выполнять команды, как если бы вы осуществляли набор с клавиатуры местного терминала на этом узле. SETHOST.EXE является одной такой утилитой для выполнения этой задачи. Командный терминал (Command Terminal — CTERM) является протоколом DECnet, используемым с SETHOST, и сравним с TELNET набора протоколов TCP/IP. Отметим, что местные средства передачи (Local Area Transport — LAT) также могут быть использованы с SETHOST. LAT не является протоколом DECnet.

- Управление сетью. — Утилиты управления и сопровождения — такие, как программа управления сетью фирмы DEC (Network Control Program — NCP) — позволяет контролировать сеть, управлять ею и проверять ее активность, а также управлять базами данных узловых определений.

- Разделение, или совместное использование, сетевых ресурсов. — Виртуальные утилиты устройств, такие, как сетевая утилита устройства (Network Device Utility — NDU) "DECnet-DOS" позволяет вам осуществлять доступ к удаленным дисководам, как если бы они были местными устройствами DOS.

PATHWORKS и DECnet

В среде PATHWORKS типичная среда DECnet одноуровневого типа включает в себя копирование файлов со стороны клиента DECnet-DOS от другого клиента DECnet-DOS с помощью сетевой передачи файлов (Network File Transfer — NFT). Примером реализации связи "клиент-сервер" в DECnet является привязка клиента PATHWORKS V5.1 к совместному использованию LAN Manager с DECnet.

Клиент DOS и сервер OpenVMS принимают по умолчанию значение кадров Ethernet II для поддержки DECnet. Клиент DOS PATHWORKS V5 поддерживает для работы стека DECnet или *свой* (*native*) канал данных, NDIS, или канал связи, соответствующий ODI. Фирма DEC допускает также конфигурацию для канала данных, совместимого с управлением каналом данных (Data Link Control — DLC), однако большинство драйверных компонент для этого канала данных должно быть получено с дискет программы поддержки локальных сетей персональных компьютеров IBM. Свой канал данных подразумевает канал данных, специфичный для продавца — фирмы DEC — с использованием DEPCA фирмы DEC или серии плат сетевого адаптера EtherWorks.

Стек DECnet у клиента DOS V5.1 реализуется принципиально одной программой TSR, известной под названием программы сетевого процесса DECnet (DNP.EXE). Этот модуль реализует сеансовые/транспортные интерфейсы DECnet, наряду с интерфейсом управления сеансом NetBIOS, специфичным для DECnet.

Механизм "от гнезда к гнезду" сети DECnet используется в качестве *внутреннего* (*underlying*) механизма связи. NetBIOS организует управление сеансом, однако транспортные средства DECnet действительно реализуют сеанс более низкого уровня через связь "от задачи к задаче", "от гнезда к гнезду" во взаимоотношениях "клиент-сервер". Программа сервера стартует первой, создает гнездо с именем или номером объекта, присвоенным гнезду, а затем пассивно ожидает возможных вызовов связи со стороны клиента.

Программа клиента DOS должна создать свое собственное гнездо, затем осуществить запрос на логическое соединение с гнездом сервера. Если сервер принимает запрос на соединение, он устанавливает логическую связь и создает другое гнездо для обмена данными с клиентом. Первоначальное гнездо остается открытым и "слушает" другие входящие запросы на связь. Это очень похоже на среду TCP/IP при программировании гнезд.

Сервер, который открывает другое гнездо для обработки обмена данными с клиентом, при использовании первоначального гнезда для прослушивания входящих запросов на связь, называется *параллельным сервером* (*concurrent server*). В случае сервера UNIX процесс родительского сервера осуществляет вызов *вилочной* (*fork*) системы для инициирования дочернего процесса для обработки обмена данными с клиентом после установления канала.

Заключение

В этой книге рассматриваются три сетевые операционные системы: NetWare, LAN Manager и PATHWORKS.
Сетевая операционная система NetWare использует набор протоколов IPX/SPX в качестве средства обеспечения своих собственных каналов связи. Этот набор протоколов представляет собой адаптированный набор протоколов Xerox Network Systems. NetWare V4.x поддерживает интерфейс X.500 Directory Services, который лучше подходит для управления сетью, чем виды сервиса редактора связей V3.x.

LAN Manager использует протокол SMB для обеспечения реализации своих функций из прикладного уровня, а NetBIOS API обеспечивает интерфейс сеансового уровня над поддерживаемыми средствами связи, такими, как TCP/IP, DECnet и расширенный пользовательский интерфейс NetBIOS (NetBEUI); NetBEUI используется с LAN Manager и Windows фирмы Microsoft для персональных компьютеров рабочих групп, а также сетевых операционных систем персональных компьютеров фирмы IBM.

Сетевая операционная система PATHWORKS фирмы Digital Equipment Corporation реализует протокол сетевой операционной системы MS-NET в версиях до V4.x. Она использует функциональность LAN Manager в его продуктах V5 и серверах OSF/1, а также у его клиентов DOS и Windows.

Хотя мы не рассматривали транспортные средства NetBEUI, они представляют собой только LAN-протокол, а TCP/IP и DECnet являются трассируемыми протоколами. NetBEUI использует кадры IEEE 802.2 и NetBIOS SAP (F0 шестнадцатиричный) для передачи данных. SAP действует как оконечный почтовый ящик и используется для идентификации конкретных протоколов. NetBEUI является транспортным протоколом ОС LAN Manager.

Фаза IV DECNet фирмы DEC использует кадры Ethernet II для передачи данных, а стек клиента TCP/IP фирмы DEC использует либо кадры Ethernet II, либо кадры IEEE 802.2. Протокол доступа к подсети (Subnetwork Access Protocol — SNAP) является адаптацией механизма 802.2 LLC SAP.

Глава 2

Взаимодействие процессов: гнезда, вызовы удаленных процедур и именованные программные каналы

МЕТОДОЛОГИЯ ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ

Введение

Программы, демонстрируемые в этой книге, используют методологию модели взаимодействия процессов, или межпрограммного взаимодействия, (InterProcess Communications — IPC) операционной системы UNIX. Гнезда (sockets) и сетевые именованные программные каналы (pipes) — это два механизма взаимодействия процессов. Они демонстрируются в этой книге с примерами программ. Вызовы удаленных процедур (Remote Procedure Calls — RPC) представляют собой другую методологию IPC, однако, в этой книге они не демонстрируются. Кроме того, в книге рассматриваются и демонстрируются гнезда Windows (Windows Sockets — WINSOCK). Взаимодействие процессов является основой для работы в сети системы "клиент-сервер".

Взаимодействие процессов: Определение/Описание

Взаимодействие процессов (IPC) — это термин, охватывающий связь между процессами в одной и той же или различных системах главных компьютеров (хостов) с помощью одного из нескольких механизмов. Таких механизмов три — это гнезда операционной системы (ОС) UNIX фирмы Berkeley Software Distribution (BSD), вызовы удаленных процедур и сетевые именованные каналы (пайпы).

Средства обеспечения взаимодействия процессов UNIX, применимые для других операционных систем, разрабатывались с учетом необходимости обеспечения прозрачности и совместимости. Прозрачность означает, что взаимодействие процессов не должно ограничиваться принадлежностью процессов к одной и той же машине. Совместимость здесь подразумевается в том смысле, что существующие процедуры обеспечения взаимодействия процессов должны использоваться и в распределенной среде "клиент-сервер" без изменений или с внесением незначительных изменений. Примером совместимости может служить тот факт, что интерфейс вызова гнезда является постоянным на всех платформах, удовлетворяющих требованиям IPC.

Реализация IPC

Для реализации вышеупомянутых целей прозрачности и совместимости было разработано несколько ключевых компонент. Первая из них — это понятие коммуникационного домена (communications domain). Система должна поддерживать коммуникационные сети, которые используют для передачи данных различные наборы протоколов, различные условные обозначения и т.д.

Термин "домен" (domain) в этом смысле нельзя смешивать с доменом LAN Manager или доменом BIND (Berkeley Internet Name Domain). Коммуникационный (связной) домен в этом смысле скорее похож на набор коммуникационных (связных) протоколов, или на семейство протоколов. Это понятие реализует стандартизованный набор семантик для связи и присвоения имен.

Примеры имен коммуникационных доменов:

- DARPA Internet — Protocol Family_Internet (PF_INET), или, в другой номенклатуре, Address Family_Internet (AF_INET). Для связи между этими двумя условными наименованиями обычно используются операторы *#define*.
- Архитектура XNS (Xerox Network System) (PF_NS).
- Сетевая архитектура IBM Systems (IBM Systems Network Architecture — SNA) (PF_SNA).
- DECnet (PF_DECnet).
- Сеть AppleTalk (PF_APPLETALK).
- Сетевой стандарт сетей с пакетной коммутацией CCITT. X.25 (AF_X25).

Вышеприведенные обозначения, например, AF_NET для Address Family_Internet, в действительности представляют собой значение поля с системным вызовом *гнезда* (*socket()*).

Следующим важным компонентом является абстракция понятия "гнезда", которым можно манипулировать с помощью дескриптора файлов. Гнездо (socket) — это абстрактный конечный пункт, с которого сообщения передаются и принимаются. Гнезда создаются внутри коммуникационного домена, подобно тому как файлы создаются внутри системы файлов. В отличие от файлов гнезда существуют только до тех пор, пока к ним имеется обращение.

Когда осуществляется системный вызов гнезда *socket()* для открывания или создания нового гнезда, программисту возвращается номер дескриптора файлов, с помощью которого гнездо идентифицируется. Этот дескриптор файлов работает таким же образом, каким он работает, например, с вызовом *open()* ОС UNIX.

Третьим из вышеупомянутых компонент обеспечения взаимодействия процессов является тот факт, что семантика присвоения имен должна быть постоянной

на всех коммуникационных доменах. Например, методы обеспечения связи, такие, как поточные гнезда (stream sockets) и гнезда дейтаграмм, или дейтаграммные гнезда (datagram sockets), должны быть доступными в виде программных запросов независимо от типа домена или типа семейства адресов.

Последним компонентом является тот факт, что гнезда должны быть именованы. "*Именованы*" (*named*) в этом контексте означает, что адреса гнезд, или номера портов, связаны со специальным протоколом, таким, как TCP, TELNET и т.д. Адреса хорошо известных портов — тех протоколов, которые резервируются операционной системой — могут быть найдены поиском их имен в базе данных. Другие адреса выбираются или кодируются программистом или присваиваются ядром UNIX из подходящего диапазона номеров.

ВЗАИМОДЕЙСТВИЕ ГНЕЗДОВЫХ ПРОЦЕССОВ

Обзор программирования гнезд

Для того чтобы писать гнездовые программы (socket programs) TCP/IP или DECnet в среде персональных компьютеров, вам необходимо иметь следующее:

- Компилятор языка C или C++ — такой, как Borland C++ — для персонального компьютера клиента, а также компилятор C в конкретной операционной системе сервера. Например, фирма DEC предлагает свой собственный компилятор C для OpenVMS, в то время как Microsoft, IBM, Novell и другие производители продают компиляторы для клиентов и серверов систем DOS и NTAS LAN Manager, OS/2 и NetWare, соответственно.
- Доступ к библиотекам гнезд у клиента и сервера. Они обычно поставляются, когда вы покупаете поддержку транспортных средств TCP/IP или DECnet и/или набор программиста, который обычно называется комплектом средств разработчика программного обеспечения (Software Developers' Kit — SDK) — для клиентских персональных компьютеров и серверов для операционных систем любого типа. Например, клиент PATHWORKS V5.1 может быть конфигурирован для программной поддержки гнезд со своей утилитой PWSETUP или пиктограммой Windows Configure при условии, что вы покупаете комплект средств разработчика программного обеспечения для установки в серверной сервисной части клиентской системы. С OS/2 поддержка программирования TCP/IP обеспечивается с помощью программного обеспечения OS/2 LAN Manager и связанных с ним SDK. Отметим, что все упомянутые библиотеки гнезд TCP/IP обычно совместимы с V4.3 BSD.

Гнезда Berkeley являются одной из форм IPC между процессами в различных хостовых системах в локальной компьютерной сети. При рассмотрении гнезд мы исследуем связь между процессом клиента и процессом сервера OSF/1 или ULTRIX в контексте набора продуктов PATHWORKS. Отметим, что вызовы библиотеки

и терминология, упомянутая здесь, должны относиться к большинству любых серверных систем TCP/IP.

Операционная система OSF/1 в версии 2.0 (когда писалась эта книга) имеет сетевой интерфейс гнезд, совместимый с версиями V4.3 и V4.4 BSD. Отметим, что она также имеет сетевой API интерфейса транспортного уровня (Transport Layer Interface — TLI) системы V. Таким образом, она по умолчанию поддерживает домены связного протокола UNIX и Internet.

Вы можете добавлять другие домены к поддержке ядра OSF/1 при помощи системного программирования. За дальнейшей информацией по этому вопросу обращайтесь к соответствующему представителю фирмы DEC. Первая реализация ОС UNIX фирмы DEC под названием ULTRIX также поддерживает эти домены протоколов, а также XNS. Последней серверной версией на системах ULTRIX во время написания этой книги была ОС MS-NET PATHWORKS для ULTRIX V1.3.

Эти наборы протоколов поддерживаются через API языка С в системе OSF/1. Внутренние средства связи, обеспечиваемые этими доменами, включают в себя гнездовую связь.

У клиента TCP/IP версии 5.1 DOS фирма DEC в настоящее время поддерживает свой собственный стек TCP/IP и гнезда BSD, совместимые с Berkeley 4.3 с незначительными модификациями. Этот гнездовой интерфейс клиента PATHWORKS поддерживает два домена протоколов:

- Домен протоколов Internet. Address Family_Internet (AF_INET) может быть определен как константа, представляющая семейство протоколов с системным вызовом гнезда.
- Домен DECnet Фазы IV для DOS/Windows. AF_DECnet определяется как константа, представляющая семейство протоколов с системным вызовом гнезда.

Перед тем как продемонстрировать пример в языке С программирования гнезд TCP в следующем разделе, мы должны рассмотреть некоторые основные принципы IPC с объяснением этих принципов:

- Гнездо представляет собой основной строительный блок для связи в среде программирования взаимодействия процессов.

  Пояснение: Гнезда являются оконечными пунктами связи между процессами в различных хостовых системах при программировании в доменах Internet или Xerox NS, а также DECnet.

  На рис. 2-1 показана сущность связи гнезд в межсетевой среде (Internet) с использованием TCP.

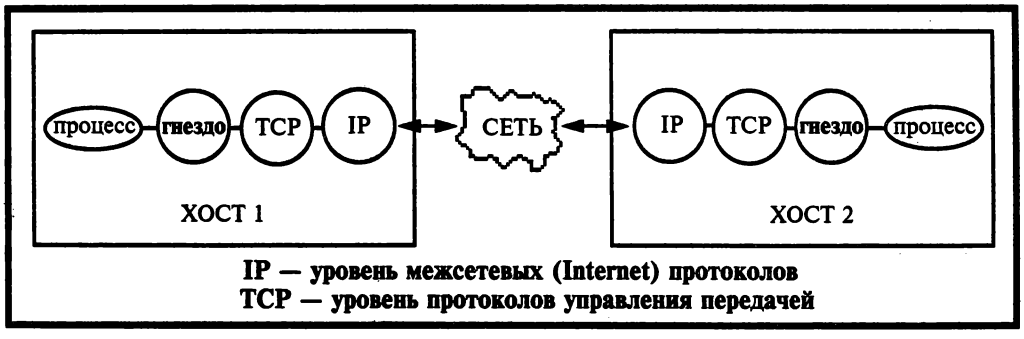


Рис. 2-1. Связь гнезд в межсетевой среде с использованием TCP.

В сервере ULTRIX или OSF/1 или любой системе BSD UNIX структура гнездовых данных в ядре является центром деятельности, когда гнездовые прикладные программы клиента-сервера связываются между собой. Стандартные подпрограммы интерфейса системного вызова управляют действиями, относящимися к системному вызову, такому, как вызов гнезда (*socket()*), собирая параметры вызова и преобразуя программные данные в формат, ожидаемый интерфейсом гнездового уровня.

Стандартные подпрограммы этого уровня непосредственно управляют структурами гнездовых данных, а также синхронизацией асинхронных, или неблокирующихся, действий. Ядро обеспечивает вспомогательные виды сервиса, таких, как управление буфером и маршрутирование сообщений, а также интерфейсы с сетевыми интерфейсными драйверами для использования различных линейных сетевых протоколов, таких, как Ethernet и Token Ring и т.д.

- Установление связи в программной среде IPC является *асимметричным* (*asymmetric*).

  Программы клиента и сервера выполняют разные роли в установлении соединения. Программирование гнезд в сетевой среде TCP/IP затрагивает интерфейс программирования клиента — сервера для IPC. Для того чтобы два процесса могли связаться друг с другом с разных сетевых машин, одна из машин должна работать как сервер, который пассивно ожидает входящие соединения, в то время как другая действует как клиент, который активно инициирует или запрашивает соединение. Сервер должен стартовать первым, затем в работу включается программа клиента.

- Сетевое соединение в среде IPC может быть контактным, или на основе логического соединения, (connection-based) или бесконтактным, т.е. без установления логического соединения (connectionless).

В сценарии на основе логического соединения клиент открывает соединение с ожидающим сервером обычно с помощью поточного гнезда TCP. Сетевой ввод/вывод (I/O) на этом соединении всегда находится между теми же самыми двумя процессами одного уровня. При протоколе без логического соединения сервер обрабатывает каждую *дейтаграмму* независимо одну от другой. *Дейтаграмма* (*datagram*) — это сетевое сообщение, которое содержит всю информацию, требуемую для доставки, и которое передается независимо от всех других сообщений. Дейтаграммы без логического соединения часто используют широковещательный протокол дейтаграммы пользователя (User Datagram Protocol — UDP) для передачи по сети.

В среде без логического соединения нет смысла в установлении соединения, так как каждый сетевой процесс входа/выхода — т.е. каждый переданный и/или принятый пакет — может иметь место между различными процессами на разных хостовых системах.

На рис. 2-2 представлены пути доступа сетевых данных при гнездовых соединениях TCP/IP.

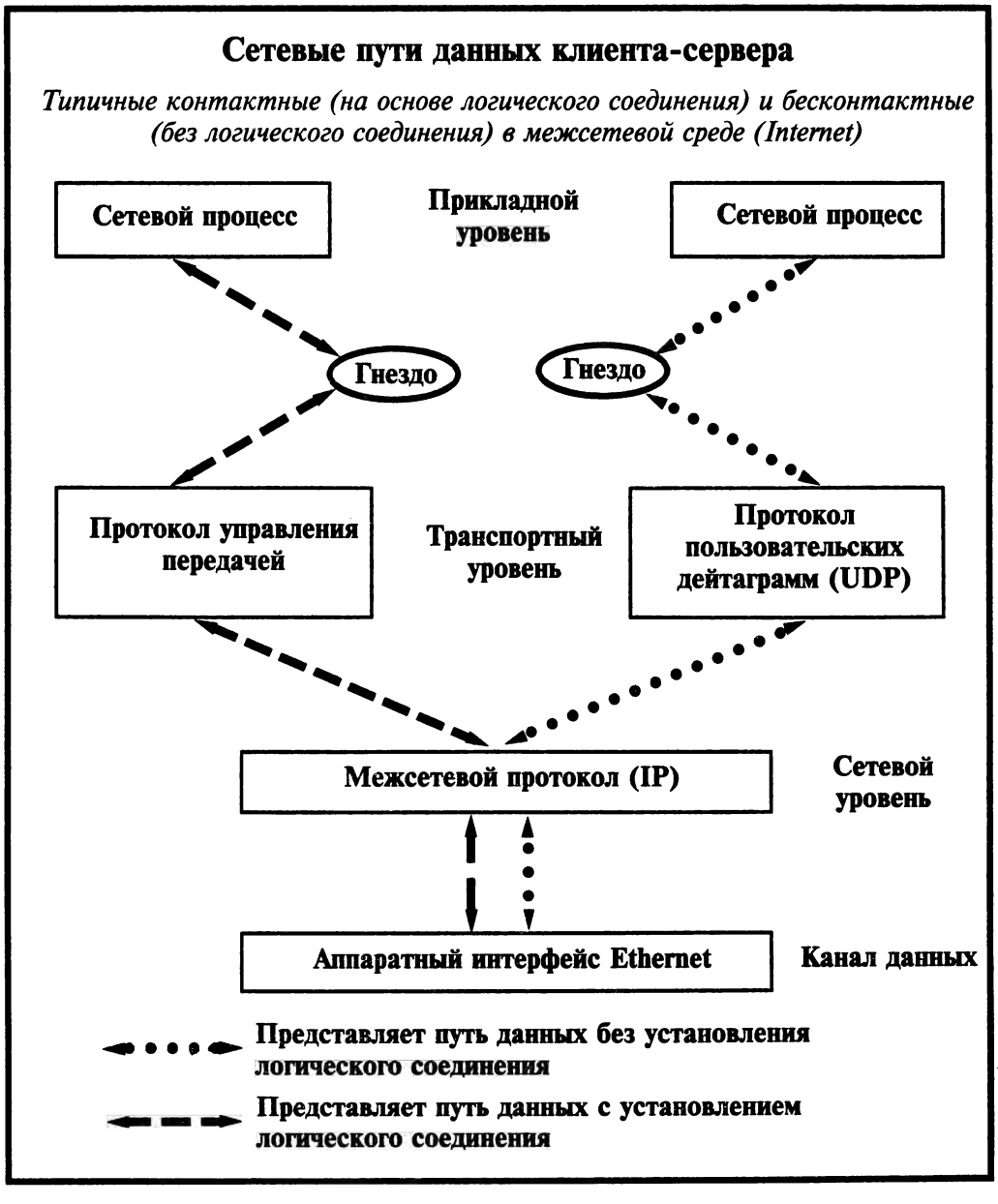


Рис. 2-2. Пути доступа сетевых данных при гнездовых соединениях TCP/IP.

- *Ассоциация* (*association*) описывает "*пятерку*" (5-*tuple*). Для осуществления успешного соединения должны выполняться требования, предъявляемые к "пятерке".

"Пятерка" представляет собой пять компонент, которые требуются для полного описания двух процессов, участвующих в успешном уникальном соединении между двумя оконечными гнездами. Элементы "пятерки" таковы:

(protocol, local-addr, local-process (port), foreign-addr, foreign-process).

Протокольная часть "пятерки" определяет протокол, такой, как TCP, который является членом специфичного семейства протоколов — домена протоколов — который вы используете. Примером семейства протоколов является семейство Internet или семейство DECnet. Серверы OSF/1 и ULTRIX поддерживают несколько семейств, как и большинство реализаций UNIX; версия PATHWORKS V5.1 для клиента TCP/IP поддерживает семейство протоколов AF_NET (Internet) и AF_DECnet (DECnet). Аргумент протокола используется с системным вызовом гнезд и обычно устанавливается в нуль в сервере UNIX, за некоторыми исключениями. Это по умолчанию переводит протокол в TCP. На рис. 2-3 показано поле аргументов протокола — "константа"у клиента PATHWORKS V5.0A DOS и его возможные значения.

Константа	Семейство	Значение	Описание
DNPROTO_NSP	AF_DECNET	0 × 01	Протокол DECnet Фаза IV для DOS/ Windows
PROTO_UDP	AF_INET	0 × 11	Протокол IP UDP
PROTO_TCP	AF_INET	0 × 06	Протокол IP TCP
PROTO_ICMP	AF_INET	0 × 01	Протокол IP RAW

Рис. 2-3. Поле констант у клиента PATHWORKS V5 DOS.

Упоминание о системном вызове гнезда (*socket()*) затрагивает интересный момент. В системах DOS и OS/2 интерфейс гнезд реализуется в виде библиотек и сетевых компонент TSR, а не в виде внутреннего компонента операционных систем, как это имеет место при реализациях UNIX и OSF/1. Таким образом, не существует стандартной библиотеки гнезд для DOS и OS/2. Каждая фирма-производитель реализует несколько отличный от других интерфейс гнезд. Это создает проблемы для продавцов, которые хотят поставлять стандартизованные средства на основе гнезд, такие, как протокол передачи файлов (File Transfer Protocol — FTP) и TELNET, протокол эмуляции терминала в платформах DOS и Windows.

Интерфейс гнезд и стека TCP/IP фирмы Microsoft первоначально был разработан фирмами Hewlett Packard и 3Com Corporation. Для DOS эта библиотека — DOS_SOCK.LIB. В последних версиях имелось две библиотеки динамических связей (Dynamic Link Library — DLL), доступных для Windows: WIN_SOCK.DLL и WSOCKETS.DLL. В настоящее время API WIN_SOCK выходит из пользования. Фирма Microsoft в настоящее время использует библиотеку WINSOCK.DLL.

В ОС PATHWORKS для клиента DOS V5.0A существует ряд файлов библиотеки гнезд, с которыми можно установить связь, в зависимости от используемой модели программирования. На рис. 2-4 показаны эти незначительные отличия.

Библиотечный файл	Описание
SPWSOCK.LIB	Малая версия для DOS, пригодная для компоновки в C
MPWSOCK.LIB	Средняя версия для DOS, пригодная для компоновки в C
CPWSOCK.LIB	Компактная версия для DOS, пригодная для компоновки в C
LPWSOCK.LIB	Большая версия для DOS, пригодная для компоновки в C
PWSOCK.DLL	DLL библиотеки гнезд для MS-Windows и Windows NT
XPWSOCKP.LIB	Версия DOS Extender для расширителей Phar Lap 386
XPWSOCKR.LIB	Версия DOS Extender для расширителей Rational DOS 16/M
XPWSOCKD.LIB	Версия DOS Extender для серверов семейства DPMI

Рис. 2-4. Библиотеки гнезд в PATHWORKS для DOS V5.0A.

PATHWORKS также обеспечивает WINSOCK.DLL со своим SDK, который совместим с WINSOCK V1.1. WINSOCK.DLL является промышленным стандартом API для гнезд Windows. WSOCKETS.DLL является более ранней собственной разработкой DLL фирмы 3Com для гнезд Windows и поддерживает интерфейс гнезд клиента Digital V2.0 TCP/IP. Последние версии TCP/IP фирмы 3Com теперь поддерживают WINSOCK.DLL. Версия PATHWORKS V5.0A библиотеки WINSOCK.DLL поддерживает только стек TCP/IP, который идет с PATHWORKS V5.0A.

При рассмотрении "пятерки" можно заметить, что компоненты *local-addr* и *foreign-addr* являются межсетевым (Internet) адресом — в формате Network ID и Host ID — в том виде, в каком они определены набором протоколов Internet (Internet protocol

suite). Например, 128.45.71.199 является 32-битовым адресом Internet класса B, выраженным в точечной нотации. Эта нотация разделяет точкой каждый байт адреса.

Части "пятерки" *local-process* и *foreign-process* представляют собой конкретные процессы на каждом из хостов, участвующих в соединении. Эти процессы на самом деле представляют собой номера портов, которые известны как *хорошо известные адреса* (*well-known addresses*), если порты представляют такой вид сервиса, как FTP или TELNET, которые может предложить хост сервера. Эти протоколы прикладного уровня являются стандартными на BSD-реализациях UNIX.

Пример полной "пятерки" на машине BSD UNIX может иметь следующий вид:

(IPPROTO_TCP,128.45.71.13, 32534, 128.45.71.38, 23)

Поле *IPPROTO_TCP* обычно устанавливается в 0. Ядро назначает *IPPROTO_TCP* типам протоколов *SOCK_STREAM*, определенным в вызове *гнезда* (*socket*).

32534 — произвольный номер порта, назначенный на стороне клиента.

Хорошо известный адрес — 23 — является номером порта для протокола TELNET.

Например, в ULTRIX некоторые из системных сетевых вызовов требуют указателя к адресной структуре гнезд, как это определено заголовочным файлом (header file) <sys/socket.h> более поздней версии ULTRIX:

```
struct sockaddr {
    u_short sa_family;      /* address family */
    char sa_data[14];       /* up to 14 bytes of direct address */
```

14 байт вышеупомянутого "непосредственного адреса" зависят от семейства используемых протоколов. Таким образом, если используется семейство Internet (AF_INET), то применяются следующие структуры из <netinet/in.h>:

```
/*
* Internet address (a structure for historical reasons)
*/
struct in_addr {
        union {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
                struct { u_short s_w1,s_w2; } S_un_w;
                u_long S_addr;

        } S_un;
#define s_addr   S_un.S_addr        /* can be used for most tcp & ip code */
```

```
    #define s_host   S_un.S_un_b.s_b2   /* host on imp */
    #define s_net    S_un.S_un_b.s_b1   /* network */
    #define s_imp    S_un.S_un_w.s_w2   /* imp */
    #define s_impno  S_un.S_un_b.s_b4   /* imp # */
    #define s_lh     S_un.S_un_b.s_b3   /* logical host */
};

    /*
    * Socket address, internet style.
    */

    struct sockaddr_in {
      short      sin_family;          /* AF_INET protocol family */
      u_short    sin_port;            /* 16-bit port number */
      struct     in_addr    sin_addr; /* 32-bit network no. / host no. */
      char       sin_zero[8];         /* unused */
}
```

В первой структуре in_addr, переменная s_addr — это число без знака (32 бита) в виде сетевого номера нумерованного хоста, с упорядоченными сетевыми байтами.

- Сетевой протокольный уровень обрабатывает "синдром" *big endian*, *little endian*.

  Термин "с упорядоченными сетевыми байтами" означает, что сетевой протокольный уровень (например, протокола TCP/IP) должен определить, использует ли он формат большого эндиана (big endian) или малого эндиана (little endian) для хранения в памяти многобайтовых адресов и других полей данных. TCP/IP использует большой эндиан для хранения в памяти 16- и 32-битовых целых чисел в своих протокольных заголовках.

  Большой эндиан означает, например, что старший байт 16-битового поля данных хранится в памяти по нижнему адресу памяти, а за ним идет младший байт в следующей старшей ячейке памяти.

  На рис. 2-5 показаны в сравнении большой эндиан и малый эндиан.

  Следует помнить, что сам протокольный уровень не управляет порядком, в котором программное обеспечение прикладного уровня передает его данные.

  Порядок сетевых байтов является средством решения проблем, возникающих из-за различных методов хранения этих многобайтовых полей, предлагаемых различными компьютерными архитектурами. Напимер, компьютер VAX фирмы DEC хранит 16-битовое целое число без знака в формате "младший-старший", в то время как некоторые машины фирмы IBM используют большой эндиан. К сожалению, стандарт на хранение данных в машинной архитектуре отсутствует. Таким образом, TCP/IP определяет, что он использует большой эндиан для решения этих проблем.

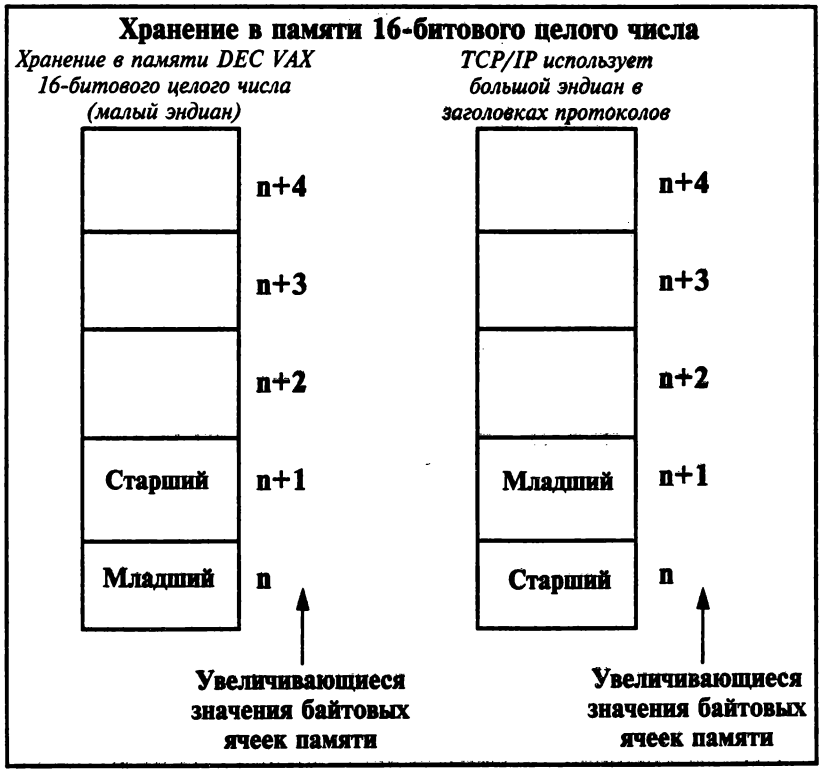


Рис. 2-5. Большой эндиан и малый эндиан.

- Вызовы, связанные с гнездами, используют или блокирующие, или неблокирующие модальности для обратного контроля.

  Например, функция *recv()* общая для сред PATHWORKS V5.1 и Microsoft TCP/IP — используется для приема данных от гнезда.

  Этот вызов по умолчанию относится к блокирующему режиму, если на гнезде нет никаких данных. Он ожидает поступления данных до возвращения управления на программу. Однако режим гнезда может быть установлен блокирующим или неблокирующим в зависимости от того, как программист хочет обрабатывать гнездовые данные.

  Например, функции PATHWORKS V5 позволяют вам использовать вызов *sioctl()* для управления рабочими характеристиками гнезда. Вы можете использовать аргумент запроса FIONBIO, затем точку с аргументом *argp*, на байт, который содержит значение либо 0 (блокирующий режим), либо 1 (неблокирующий режим) для установки этой характеристики гнезда.

Вызов *sioctl()* PATHWORKS V5 принимает следующий вид:

```
int sioctl(s,request,argp)
        int s,request
        unsigned char *argp;
```

Когда мы переносим нашу программу гнезд клиента DEC V2.0 TCP для работы под PATHWORKS V5, мы определяем argp как:

```
long far *argp
```

для вызова *ioctl()*, в противоположность *sioctl()* ОС PATHWORKS V5. Таким образом, для V5 мы переводим это в вид:

```
if(sioctl(sockfd,FIONREAD,(unsigned char *)argp))
.
.
.
```

Отметим, что мы не использовали *sioctl()* для установки гнезда в блокирующий или неблокирующий режим; мы использовали его в этом случае для определения количества байтов данных, имеющихся на гнезде для считывания (запрос FIONREAD). Другими функциями гнезда, которые могут осуществлять блокирование, являются *accept(), connect (), recvfrom(), send(), sendto()* с библиотеками TCP/IP Microsoft и *accept(), connect(), sread(), send(), swrite()* для PATHWORKS V5.

ВЫЗОВЫ ГНЕЗД

Введение

С учетом вышеупомянутого рассмотрим ключевые системные вызовы, используемые при программировании гнезд IPC на серверах ULTRIX и OSF/1 и клиенте с персональным компьютером с ОС PATHWORKS V5. Позже мы сравним вызовы программы клиента с вызовами, которые нужны для работы на реализации TCP/IP/DOS, а также с теми, которые необходимы для работы системы Windows NT. Эти вызовы очень похожи друг на друга.

Здесь будут рассмотрены не все различия между реализациями UNIX BSD и клиентами PATHWORKS или LAN Manager. Для получения полных подробностей прочитайте соответствующую документацию.

Отметим, что справочник-учебник по PATHWORKS называется "*PATHWORKS Network Programming Reference*", номер издания (part number) AA-PAFJD-TK. Этот учебник является необходимым для программирования клиентов PATHWORKS,

TCP/IP и DECnet, и он включен в комплект распределения программного обеспечения PATHWORKS (Software Distribution Kit — SDK), который можно заказать у фирмы DEC. Документация Microsoft имеется вместе с ее SDK для LAN Manager/TCP/IP (в момент написания этой книги номер издания — 0962744-005).

Вызовы гнезд

- ***socket()***

Системный вызов гнезда используется для создания гнезда и требует следующего формата:

socket(af,type,protocol);

где *af* обозначает семейство адресов (или протокольный домен); *type* означает тип протокола, например *SOCK_STREAM* для гнезда поточного типа; и *protocol* обычно устанавливается в 0 для значения *IPPROTO_TCP* по умолчанию или TCP в BSD UNIX.

На рис. 2-6 приведены типы и протоколы, которые могут быть использованы с семейством адресов домена Internet, или AF_INET, на сервере ULTRIX, совместимом с V4.3 BSD.

Постоянная *типа* (*type*) протокола	***Протокол (protocol)***	**Реализация протокола**
SOCK_DGRAM	IPPROTO_UDP	UDP
SOCK_STREAM	IPPROTO_TCP	TCP
SOCK_RAW	IPPROTO_ICMP	ICMP
SOCK_RAW	IPPROTO_RAW	RAW

Рис. 2-6. Типы и протоколы, которые могут быть использованы с семейством адресов домена Internet.

Поточное гнездо — это гнездо, на котором имеет место двунаправленный, надежный, упорядоченный и недублированный поток данных. У клиентов PATHWORKS V5.1 и LAN Manager V2.2 TCP/IP вы определяете одну и ту же постоянную типа гнезда (*type*) — SOCK_STREAM.

Вызов *socket()* имеет тот же вид у клиента, при этом разрешаются незначительные исключения относительно констант протокола и типа. Однако существует одно

большое различие в реализации файловых и гнездовых дескрипторов на сервере ULTRIX или любом BSD-совместимом сервере по сравнению с персональным компьютером.

При работе под ОС ULTRIX или OSF/1 файловые и гнездовые дескрипторы распределяются из одного и того же пространства целочисленных дескрипторов. Например, системные вызовы, такие, как *read()*, которые реализуют функции ввода/вывода на файловых дескрипторах, работают также и на гнездовых дескрипторах. В DOS и Microsoft (HP и 3Com3+Open) или программном обеспечении DEC PATHWORKS (V5.1) TCP/IP файловые дескрипторы и гнездовые дескрипторы приходят из различных пространств. Они идут от поддержки гнезд, встроенной в загружаемые сетевые TSR, относящиеся к гнездам.

Существует одна постоянная *type*, доступная для клиента PATHWORKS V5.1. Она обычно доступна, когда вы определяете семейство AF_NS (Xerox Network Systems) в системе BSD UNIX. Это SOCK_SEQPACKET, которая связана с типом протокола упорядоченных пакетов (Sequenced Packet Protocol — SPP) для семейства Xerox NS. С PATHWORKS V5.1 вы можете задать значение *type* для SOCK_SEQPACKET, равное 0x05 (шестнадцатиричное 5), которое может использоваться, когда вы используете DECnet для того, чтобы связаться с другой системой. Гнезда SOCK_SEQPACKET аналогичны дейтаграммам, за исключением того, что гарантируется их принятие в той последовательности, в которой они были посланы. Также гарантируется их безошибочность.

- ***bind()***

Система привязывания (bind system) привязывает имя к объекту без имени. Этот вызов регистрирует адрес TCP/IP и номер порта со структурой адреса гнезда для конкретного семейства протоколов. Номер порта может быть хорошо известным адресом или числом, произвольно заданным ядром. Он связан с номером гнездового дескриптора (или номером файлового дескриптора), возвращенным с гнездовым вызовом. В действительности этот вызов говорит операционной системе — или резидентному модулю "завершить-и-остаться в памяти" (TSR), который реализует гнезда в системе DOS — что любые сетевые пакеты для этого адреса или порта должны передаваться на серверную прикладную программу, которая создала этот пакет.

Вызов *bind* позволяет процессу определить половину ассоциации, т.е. определяются локальный адрес и локальный процесс — {local-addr, localprocess (port)}. Вызов *socket()* определяет протокол, такой, как NCP, который является пятым элементом "пятерки". Другими двумя являются *foreign addr* и *foreign process.*

Вызов привязывания имеет вид:

```
bind (sockfd, name, namelen);
```

где *sockfd* номер гнездового дескриптора, возвращенный с вызовом *socket()*, *name* —

это указатель к адресной структуре конкретного протокола, а *namelen* — это размер этой адресной структуры в байтах. У клиента PATHWORKS V5.0 DOS DECnet использует *sockad dr_dn*, а TCP/IP использует *sockaddr_in*. TCP/IP LAN Manager использует *sockaddr_in* для TCP/IP.

Таким образом, вызов привязывания заполняет локальный адрес — 32-битовый адрес TCP/IP — идентификатора сети/идентификатора хоста и локальный номер порта, который должен связываться с номером дескриптора гнезда. Номер порта связан с единым семейством протоколов от вызова гнезда, который создал дескриптор. Вызов гнезда просто создает гнездо, которое существует в адресном пространстве, но которому адрес не присвоен.

При привязывании межсетевого (Internet) имени (или полуассоциации) к гнезду вызов обычно принимает следующую форму при реализации на сервере ULTRIX, OSF/1 или BSD UNIX.

```
#include    <sys/types.h>
#include    <netinet/in.h>
      .
      .
      .
struct sockaddr_in sin;
      .
      .
      .
bind (sockfd, (struct sockaddr *) &sin, sizeof (sin));
```

Вызов принимает ту же общую форму с программным обеспечением Microsoft TCP/IP (HP 3Com,3+Open), со следующими включаемыми файлами:

```
#include    <sys/socket.h>
#include    <netinet/in.h>
#include    <sys/types.h>
```

- ***listen()***

Для того чтобы сервер принял соединение с клиентом, он должен выполнить два системных вызова после привязывания гнезда. Первый вызов — это *listen()*, который используется сервером, ориентированным на логическое соединение (TCP), для указания на его желание слушать входящие запросы на соединение от клиента. Он выполняется сразу же после вызовов *socket* и *bind* и до вызова *accept*, который является вторым вызовом, подлежащим выполнению.

Сервер, ориентированный на логическое соединение, использует системный вызов *fork()* для разветвления или порождения дочернего процесса для обработки запросов *accept* и соответствующих функций ввода/вывода на соединении. Он назы-

вается *параллельным* (*concurrent*) сервером TCP. С этим методом родительский процесс может выполнять другой запрос *accept* для обработки дополнительных запросов, которые, возможно, образовали очередь или только что поступили от других клиентов.

В серверах ULTRIX и OSF/1 вызов *listen()* принимает следующий вид:

```
listen (sockfd, backlog) ,
```

где аргумент *backlog* определяет, сколько запросов на соединение может быть поставлено системой в очередь, в то время как она ожидает выполнения сервером вызова приема. Это значение обычно устанавливается равным 5. Аргумент *sockfd* возвращается из вызова *socket.*

Этот вызов принимает тот же вид у клиентов DOS Microsoft и DEC.

- ***accept()***

Вызов *accept* принимает первый пришедший в очередь запрос на соединение и создает другое гнездо с теми же самыми свойствами, что и аргумент *sockfd*, но с новым номером дескриптора файла. Первоначальное гнездо остается открытым. Это — блокирующий вызов, означающий, что вызов *блокируется (blocks)*, или ожидает. Он не возвратится к источнику вызова до тех пор, пока не будет осуществлено соединение или же пока вызов не будет прерван сигналом на процесс обработки.

Если сервер хочет избежать блокирования по этому вызову или хочет принимать соединения по нескольким гнездам, существует несколько альтернатив. Однако для простоты обсуждение первоначальных программ гнезд системы "клиент-сервер" не будет включать в себя такие альтернативы. Более подробно эта тема рассмотрена в описании многогнездовых программ системы "клиент-сервер" (MULTI Client/Server Sockets program), начинающемся на стр. 167 (Глава 5).

accept() принимает вид:

```
struct sockaddr_in client;
        .
        .
        .
 clientlen = sizeof(client);
 newsk = accept (sockfd, (struct sockaddr *) &client, &clientlen);
```

Аргументы *client* и *clientlen* возвращают адрес гнездовой структуры процесса клиента, который получает соединение. Результирующая величина параметра *clientlen* инициализируется сервером, чтобы показать, сколько буферного пространства связано с аргументом *client.* Сам системный вызов хранит в памяти результат

в *clientlen* после возвращения, который является действительным количеством байтов, хранящихся в аргументе *client*. Вследствие того, что размер адресной структуры межсетевого адреса гнезд является постоянным (16 байт), значение, хранящееся в аргументе *clientlen* является действительным результатом подсчета байтов, который системный вызов возвращает в параметре клиента.

Вызов *accept()* на компьютере осуществляет возвращение гнездовых дескрипторов, которые выделены не из пространства файловых дескрипторов.

- ***connect()***

Клиент использует вызов *connect* для установления соединения между местным процессом и "чужим" процессом (процессом сервера). Если гнездо представляет собой тип SOCK_DGRAM, этот вызов постоянно определяет ранг, по которому посылаются дейтаграммы. Если оно относится к типу SOCK_STREAM, этот вызов пытается осуществить соединение на другое гнездо. Другое гнездо определяется с помощью *addr*, который представляет собой адрес коммуникационного пространства гнезда.

Процесс клиента не должен использовать вызов *bind* для привязывания (*bind*) адреса до использования соединения; сам вызов *connect* осуществит присвоение величины *local-addr.*

Соединение вызывает назначение четырех элементов "пятерки": *local-addr, local-process, foreign-addr, foreign-process.*

connect() принимает вид:

```
rc = connect (sockfd, addr, addrlen),
```

где *addr* — это указатель на одну из структур *sockaddr* удаленного сервера — либо *sockaddr_dn* для DECnet, либо *sockaddr_in* для TCP/IP.

rc = 0, если вызов успешен. Если вызов не осуществился, rc = -1.

Если вызов осуществляется от клиента на сервер ULTRIX или OSF/1, вы можете увидеть соединение на сервере с помощью следующей команды:

```
# netstat -f inet
```

Это показывает семейство межсетевых соединений через TCP/IP, и вывод должен выглядеть примерно так:

```
Active Internet connections
Proto Recv-Q Send-Q  Local Address        Foreign Address        (state)
```

```
tcp   0    0    owlp12.alf.dec.c.6000  oiws5.alf.dec.co.44409   ESTABLISHED
      .
      .
      .
```

Это показывает, что хост owlp12 имеет установленное соединение с хостом клиента oiws5. Номер порта 6000 кодируется жестким кодом на сервере в эталонной программе. 44409 — это номер порта клиента (а не номер гнездового дескриптора), который произвольно присваивается программным обеспечением TCP/IP.

Последовательность событий в системе "клиент-сервер"

Типичные сервер и клиент поточных гнезд TCP/IP обычно придерживаются последовательности событий, изображенной на рис. 2-7 . Клиентские вызовы *read()* и *write()* на диаграмме предполагают клиента ULTRIX или OSF/1. Вызовы, доступные для ввода/вывода гнезд у клиента, рассматриваются на следующих страницах этой книги.

Когда устанавливается гнездовое соединение, программист сервера может выдать стандартные системные вызовы для осуществления ввода/вывода на файловые дескрипторы. Вы можете также заменить гнездовые дескрипторы, возвращенные от вызовов *socket()* и *accept()*, на файловые дескрипторы.

У PATHWORKS для клиента DOS V5 имеется четыре основных связанных с гнездами вызова для обработки ввода/вывода гнездам. Это: *send(), swrite(), sread(), recv()*. У клиента TCP/IP LAN Manager также имеется четыре вызова: *send(), sendto(), recv(), recvfrom(),* — которые используются, в основном, для обработки ввода/вывода гнезд.

Нормальные системные вызовы DOS для выполнения ввода/вывода на файловых описателях (*handles*), аналогичных *файловым дескрипторам* (*file descriptors*) UNIX, не могут быть использованы, так как описатели (handles) и гнездовые дескрипторы назначаются из различных пространств целых чисел. Серверы ULTRIX и OSF/1 обеспечивают те же самые четыре вызова, как и для клиента Microsoft, с незначительными изменениями.

У сервера и клиента ULTRIX *send()* принимает следующий вид:

```
#include <sys/socket.h>
.
.
.
numsent = send(sockfd, buf, len, flags);
```

У клиента DOS *buf* представляет собой дальний указатель на буфер символов, который содержит сообщение. Дальние указатели используются у клиента из-за

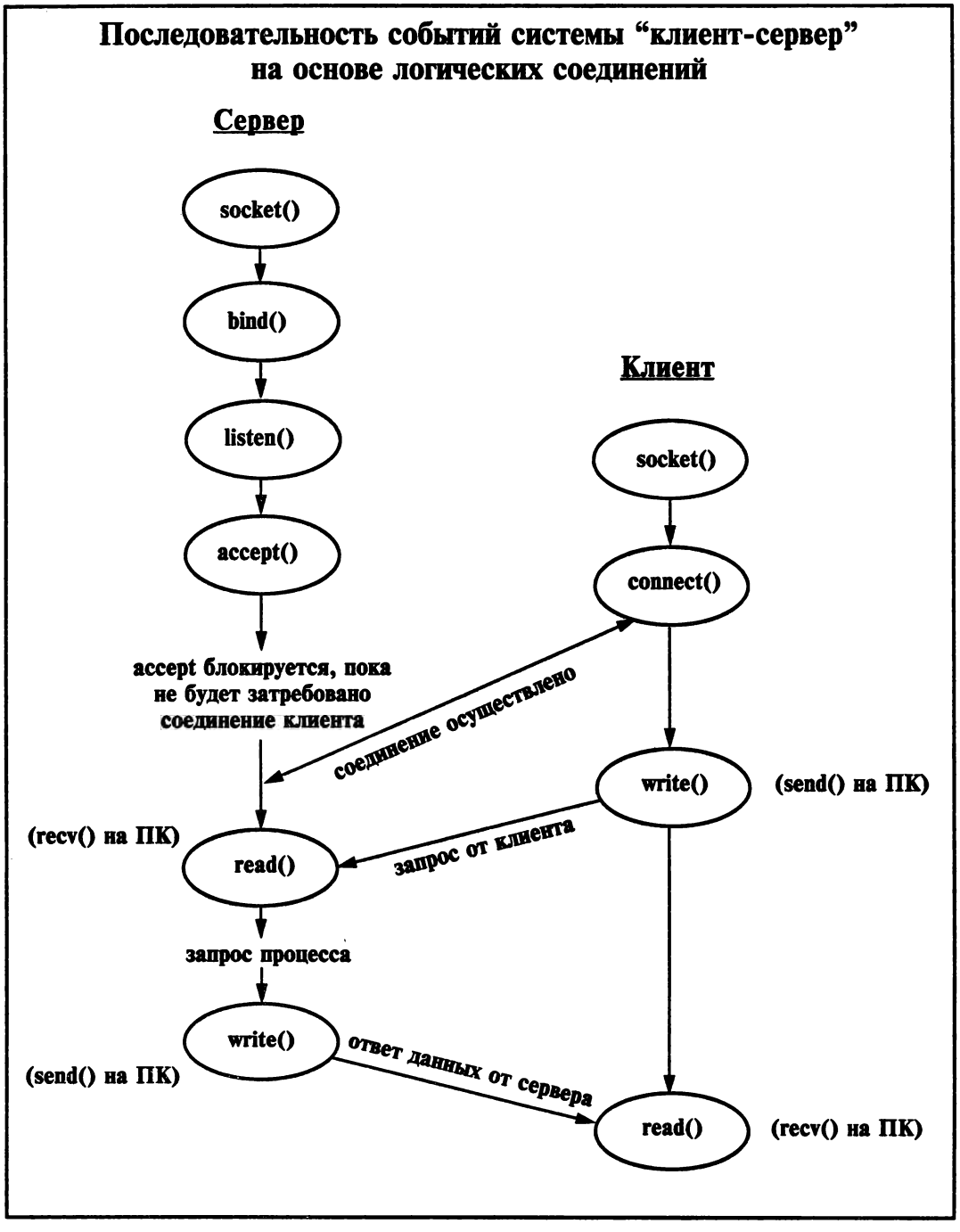


Рис. 2-7. Последовательность событий для типичного сервера и клиента поточных гнезд TCP.

сегментационной архитектуры микросхем Intel, используемых в большинстве персональных компьютеров. Дальний указатель состоит из регистровой дескрипторной пары *segment:offset_address*, такой, как регистр кодового сегмента:регистр указателя команд — Code Segment register:Instruction Pointer register (CS:IP). Пара "сегмент:регистр смещения" (segment:offset register) может указывать любой адрес в полном одномегабайтовом адресном пространстве DOS.

Число *numsent* является целым, а *len* является количеством байтов в буфере. Для гнезд SOCK_STREAM в блокирующем режиме *len* имеет максимальную величину 32767 байт. Параметр *flags* у клиента PATHWORKS может иметь значение либо 0, либо значение, выходящее из диапазона (MSG_OOB). У клиентов TCP/IP Microsoft флаги могут быть MSG_PUSH. Это — не зависящий от реализации флаг, используемый только для отсылки при поточных соединениях. Он заставляет TCP отсылать данные немедленно.

По умолчанию, операции с гнездами используют блокирующий режим. Если не оговорено другого, операция *sread()* блокирует процесс вызова до тех пор, пока данные не станут доступными для считывания. Вызов *swrite()* блокирует процесс вызывания до тех пор, пока не будет накоплено достаточно ресурсов для буферирования данных для операции записи.

При успешном выполнении *send()* возвращает количество переданных символов. При отказе этот оператор возвращает -1. Если гнездо установлено в неблокирующий режим и данные не могут успешно буферироваться к моменту выдачи *send()*, -1 возвращается с сообщением ошибки EWOULDBLOCK. *send()* является блокирующим по умолчанию.

Вызов *recv()* принимает вид:

```
#include <sys\socket.h>
.
.
.
numrecv = recv(sockfd, buf, len, flags);
```

Опять *buf* является дальним указателем на буфер приема, а *len* — это максимальное количество байт, которое вместится в буфер, указанный оператором *buf*. Значение флагов определяет, как обрабатываются данные. В соответствии с документацией, если флаг установлен в MSG_PEEK (Microsoft), все данные обрабатываются таким образом, как будто они не были считаны. MSG_OOB (DEC PATHWORKS) указывает, что запрашиваются данные, выходящие за пределы диапазона. MSG_USRBUF является сообщением для средств передачи о том, что принимаются потенциально большие сообщения, например, длиннее 4096 байт. Когда *flags* равен 0, данные считываются один раз и после этого становятся недоступными. Если *recv()* не проходит, он возвращает -1, а внешняя переменная *errno* (Microsoft) или *errval* (DEC PATHWORKS) содержит код ошибки.

Отметим, что внедиапазонные данные — это данные, которые отправитель хочет послать ранее любых данных, которые были буферированы местным сервисным уровнем байтового потока, таким, как TCP. Кроме того, отправитель хочет, чтобы приемник обрабатывал эти данные ранее любых данных, которые буферировал приемник.

Вызовы TCP/IP фирмы Microsoft *sendto()* и *recvfrom()* работают подобно *send()* и *recvfrom()*. Исключение состоит только в том, что *sendto()* позволяет уточнять адрес получателя TCP/IP каждый раз, когда осуществляется отсылка, а *recfrom()* может возвращать адрес гнезда, откуда были посланы данные. Вызов *swrite()* фирмы DEC, или "запись гнезда" ("socket write"), аналогичен ее вызову *send()* с тем отличием, что параметр *flags* не используется, что означает, что внедиапазонные данные не могут быть отосланы. Вызов *sread()* фирмы DEC аналогичен ее вызову *recv()*, с той лишь разницей, что флаги не определяются оператором *sread()*.

Единственными другими вызовами со стороны персонального компьютера, связанными с гнездами и используемыми в примерах программ, показанных далее в этой книге, являются вызовы *ioctl()* (Microsoft) и *sioctl()* (DEC PATHWORKS V5). Эти вызовы используются для манипулирования характеристиками гнезд, включая установку или сброс неблокирующих ввода/вывода, а также для определения того, сколько байтов доступны для считывания до осуществления вызова *recv()*. Например, мы используем вызов Microsoft/3Com *ioctl()* в одном из примеров программ, для того чтобы определить, сколько байтов находятся на гнезде для считывания. Он принимает такой вид:

```
#include     <sys\socket.h>
#include     <sock_err.h>
             .
             .
             .
int          sockfd;
long          retcode;
long          far *argp;
argp = &retcode;
             .
             .
             .
if (ioctl(sockfd,FIONREAD,argp))
{
  printf(" ioctl failed(%d)\n",errno);
  close_socket(sockfd);
  return;
}
```

Вызов *ioctl()* при успешном осуществлении возвращает 0. Если он не проходит, то возвращает -1. Если вызов неудачен, то во внешнюю переменную *errno* передается код ошибки. Эти величины могут быть расшифрованы с помощью просмотра включаемого файла *sock_err.h* и соответствующего справочного пособия.

Функция запроса FIONREAD запрашивает количество байтов, доступных для считывания на гнезде. Если вызов успешен, то он возвращает это число в *argp*. *FIONBIO* требует, чтобы гнездо было установлено в блокирующий или неблокирующий режим в зависимости от значения *argp*, который является дальним указателем на большое целое число. Если это целое число равно 0, то гнездо маркируется для режима блокирования. Если это число не равно нулю, то дескриптор маркируется для неблокирующего режима.

Этим мы завершаем обсуждение программных интерфейсов API для гнезд Berkeley и переходим к рассмотрению пакетов Windows.

ГНЕЗДА WINDOWS

Введение

В системах DOS и OS/2 интерфейс гнезд реализуется в виде библиотек и резидентных программ "завершить-и-остаться в памяти" прикладного уровня. Он не реализуется как внутренняя компонента операционной системы, как это имеет место при реализациях BSD UNIX и OSF/1. Таким образом, не существует стандартной библиотеки гнезд для DOS и OS/2, и стандартизация является здесь проблемой. Спецификация Windows Sockets делает попытку решить эту проблему для систем, использующих интерфейс Microsoft Windows, включая NT.

Определение/Описание: Windows Sockets

Windows Sockets (WinSock) представляет собой спецификацию фирмы Microsoft, которая определяет сетевой программный интерфейс для Microsoft Windows. Windows Sockets базируется на реализации BSD (V4.3) программирования гнезд TCP/IP. В момент написания этой книги спецификации имели версию 1.1. Однако спецификация Windows Sockets включает в себя также расширения, специфичные для Windows, которые позволяют программисту использовать механизмы Windows, управляемые сообщениями.

Спецификация Windows Sockets предназначена для использования в качестве единого стандарта, по которому разработчики могут создавать программы. Эта спецификация определяет библиотечные вызовы и связанные с ними синтаксис/семантику, по которым разработчик может создавать программы, а продавцы программного обеспечения могут их реализовывать. Спецификация Windows Sockets предназначена для использования с альтернативными протокольными стеками (помимо TCP/IP), которые ожидаются, когда будет выпущена версия V2.0 Windows Sockets. Поддержка альтернативного протокольного стека и Windows Sockets в этой книге рассматриваться не будет.

Windows Sockets облегчает перенос программ. Например, программист, который пишет прикладную программу в соответствии со стандартом WINSOCK и использует реализацию динамически компонуемой библиотеки (Dynamic Link

Library — .DLL) WINSOCK одного поставщика, должен быть в состоянии осуществлять перенос на реализацию подобной библиотеки WINSOCK другого поставщика. Теоретически эта прикладная программа должна работать с соответствующей протокольной реализацией WINSOCK.LIB и WINSOCK.DLL любого поставщика.

Перенесение прикладной программы, написанной в соответствии с WINSOCK, на платформу Windows/TCP/IP другого поставщика должно требовать в большинстве случаев перекомпиляции и перекомпоновки кода соответствующего файла WINSOCK.LIB того или иного поставщика. Если WINSOCK поставщика включает в себя библиотеку и файлы .DLL, на 100% совместимые с WINSOCK, и на 100% совместимый код прикладной программы, то теоретически не нужно осуществлять никакой перекомпиляции и перекомпоновки. Ячейка файла WINSOCK.DLL конкретного поставщика должна быть помещена в PATH для того, чтобы выполнить программу. Библиотечный файл определяет функции. Файл .DLL реализует код, который выполняет функции для сетевого транспортного уровня, например, TCP/IP и DECnet. Таким образом, вызовы WINSOCK, которые на 100 процентов совместимы со спецификациями, должны работать над любыми поддерживаемыми транспортными средствами в рамках реализации библиотечных функций .DLL.

Прикладной программе труднее поддерживать множественные гнездовые библиотеки TCP/IP для DOS. Там не существует динамической компоновки, такой, как Windows DLL. Каждая прикладная программа должна быть перекомпилирована или, по крайней мере, перекомпонована с применяемой библиотекой гнезд TCP/IP.

ПРОГРАММИРОВАНИЕ ГНЕЗД: WINDOWS И DOS

Введение

С гнездовым интерфейсом Windows сетевое программирование гнезд в среде персональных компьютеров — имеется в виду DOS и Windows — предлагает ряд альтернатив. Рассмотрим, какие библиотеки гнезд и файлы включения необходимы в конкретной среде.

Основы программирования

При создании новой гнездовой программы в языке C, написанной для операционной системы MS-DOS (вне Windows), вы обычно включаете заголовочный файл (header file), как правило, поставляемый поставщиком, в исходную программу, а затем осуществляете компоновку с библиотечным файлом, поставленным поставщиком. Так как функции, выполняемые в DOS, не поддерживают файлы библиотеки динамических связей (Dynamic Link Library — DLL), вам нужно только выбрать правильный библиотечный файл для типа модели памяти DOS, которую вы хотите создать. Так как библиотечные файлы содержат весь код, который будет введен в файл .EXE, нет необходимости в дополнительных файлах во время прогона программы.

Например, PATHWORKS V5.0A имеет несколько заголовочных файлов, которые определяют все структуры DOS, связанные с гнездами для большинства транспортных средств, включая TCP/IP и DECnet. Если вы включите основной заголовочный файл PRGPRE.H в вашу программу, тогда он будет автоматически включать в себя все другие заголовочные файлы. Вместо этого вы можете выбрать для включения только конкретные заголовочные файлы, которые вам необходимы для вашего кода. В этом случае вы должны будете проконсультироваться с таблицами в справочнике PATHWORKS Network Programming Reference для решения вопроса, какой заголовок необходимо включать.

Когда вы скомпилировали исходную программу и создали файл (файлы) .OBJ, вам будет необходимо скомпоновать все файлы .OBJ вместе с надлежащим файлом .LIB для создания вашего окончательного .EXE. Большинство поставщиков обеспечивают множественные файлы .LIB, предназначенные для работы с различными моделями памяти, доступными в MS-DOS. Например, малая модель памяти (Small Memory Model) используется, когда общий объем кода и данных превышает 64K байт, но каждая отдельная величина из этих двух — меньше 64K байт. Средняя модель памяти (Medium Memory Model) используется, когда общий объем данных меньше 64K байт, но объем программы больше 64K байт. Большая модель памяти (Large Memory Model) используется, когда объем данных превышает 64K байт и программа больше 64K байт.

Рассмотрим такой пример: для создания реализации DOS со средней моделью памяти фирма DTC обеспечивает библиотечный файл под названием библиотека PATHWORKLS Sockets средней модели (MPWSOCK.LIB), с которой следует осуществлять компоновку.

На рис.2-8 приведен список библиотечных файлов, которые фирма DEC обеспечивает для сред DOS, Windows и NT.

Библиотечный файл	Описание
SPWSOCK.LIB	Версия малой модели, компонуемой в C, для DOS
MPWSOCK.LIB	Версия средней модели, компонуемой в C, для DOS
CPWSOCK.LIB	Версия компактной модели, компонуемой в C, для DOS
LPWSOCK.LIB	Версия большой модели, компонуемой в C, для DOS
SPWSOCKW.LIB	Версия малой модели, компонуемой в C, для Windows
MPWSOCKW.LIB	Версия средней модели, компонуемой в C, для Windows
CPWSOCKW.LIB	Версия компактной модели, компонуемой в C, для Windows
LPWSOCKW.LIB	Версия большой модели, компонуемой в C, для Windows
PWSOCK.DLL	DLL-библиотеки гнезд для MS Windows и Windows NT
XPWSOCKP.LIB	Версия DOS Extender для расширителей Phar Lap 386
XPWSOCKR.LIB	Версия DOS Extender для расширителей Rational DOS 16/M
XPWSOCKD.LIB	Версия DOS Extender для семейства серверов DPMI
WINSOCK.LIB	Библиотека Windows Sockets для Windows

Рис. 2-8. Библиотечные файлы фирмы DEC для DOS, Windows и NT.

При создании программ под Microsoft Windows вы можете выбирать между использованием тех же функций гнезд, как это обеспечивается на уровне DOS, и использованием стандарта WINSOCK, разработанного фирмой Microsoft. Стандарт WINSOCK представляет собой набор специальных гнездовых функций, которые должны быть “завернуты” в DLL каждым поставщиком.

Этот стандарт дает вашей программе высокую степень переносимости (портативность) от одной платформы Windows к другой. Другим примером этого фактора портативности является тот факт, что программа WINSOCK, написанная для Windows 3.1, будет также компилироваться и работать на Windows NT. Это происходит потому, что гнезда Windows обеспечивают реализацию как для 16-битовой среды, так и для 32-битовой среды (NT).

Большинство функций WINSOCK являются теми же самыми, что и гнездовые функции Berkeley. Однако существуют расширения и некоторые тонкие различия в общих вызовах. Например, интерфейс WINSOCK включает в себя вызовы, такие, как *WSAStartup()* и *WSACleanup()*, которые являются обязательными при программировании API WINSOCK. Примерами других вызовов являются *WSAASyncSelect(), closesocket()* и *select()*. Для получения более подробной информации проконсультируйтесь со спецификациями гнезд Windows. Для получения справочного материала по гнездам Windows отправьте электронной почтой по адресу *wnsock@microdyne.com* сообщение для получения документа, озаглавленного "Windows Sockets: An Open Interface for Network Programming under Microsoft Windows".

Чтобы запрограммировать стандарт WINSOCK, вы должны включить заголовочный файл WINSOCK.H, который должен быть поставлен поставщиком. Для программирования WINSOCK поставщик также поставляет файл .LIB для осуществления компоновки и файл .DLL для его представления в момент прогона программы.

Так как WINSOCK работает в среде WINDOWS, осуществляется поддержка DLL, а программа, использующая WINSOCK, будет использовать гнезда DLL. Как программист, использующий стандарт WINSOCK, вы будете осуществлять компоновку с DLL в прозрачном режиме. Visual C++ фирмы Microsoft позволяет вам выбирать модель .DLL Windows из интерфейса меню *Project/Options.*

DLL работает, потому что код программы идет двумя частями. Файл .LIB невелик и содержит, в основном, имена функций. Файл .DLL является большим и содержит код программы для выполнения функций.

Небольшой файл .LIB введен в файл .EXE программиста. Поэтому файл .EXE не содержит всего кода программы, который необходим для выполнения программы. Когда вы запустили файл .EXE, он будет по умолчанию искать связанный файл .DDL в каталоге Windows\SYSTEM. Файл .EXE будет загружать в память код программы, находящийся там, если никакая другая выполняемая функция раньше его не загрузила. Файлы DLL являются образами памяти, которые могут совместно использоваться многими программами Windows.

PATHWORKS V5 обеспечивает файл включения WINSOCK.H и WINSOCK.DLL, который имеет информацию о TCP/IP и DECnet.

Отметим, что для поддержки DECnet WINSOCK, PATHWORKS V5 обеспечивает файл включения DWINSOCK.H, который обрабатывает определения транспортных средств, связанные с DECnet, а также библиотеки PWIMPORT.LIB и LPWLINK.LIB для компоновки.

При программировании гнезд Windows для TCP/IP в этой среде вам необходимо связаться с файлом WINSOCK.LIB. WIMNSOCK.DLL будет найден во время работы программы. С SDK PATHWORKS V5, как и с большинством реализаций WINSOCK, в заголовочном файле WINSOCK.H предусмотрены стандартные функции WINSOCK. Имена функций предусмотрены в файле WINSOCK.LIB, а действительная кодовая реализация предусмотрена в WINSOCK.DLL.

Если вы принимаете решение не использовать стандарт WINSOCK и хотите вызвать гнездовые функции подпрограммы DOS, но работать по-прежнему под Windows, вам будет необходимо использовать различные библиотеки, поставленные вашим поставщиком для обработки в этой среде программирования.

Фирма DEC позволяет вам обойти WINSOCK и перейти непосредственно к гнездовым программам DEC, определенным в *a*PWSOCL.LIB, где *a* символ ASCII, представляющий соответствующую модель памяти — S для малой, M для средней, C для компактной и L для большой. Это означает, что вы осуществляете *включение* (*#include*) PRGPRE.H и осуществляете компоновку с MPWSOCK.LIB для программы средней модели памяти, которая будет работать только под DOS. Файл MPWSOCK.LIB содержит все имена функций и код, т.е. действительный код для выполнения функций будет введен в файл .EXE.

Фирма DEC также обеспечивает — в пути каталога \PWPRGKIT\LIB (V5.0A) — библиотеки в виде *x*PWSOCK*y*.LIB, где:

- x — модель памяти — *s, m, c, l* для малой, средней, компактной или большой, соответственно;
- y имеет значение *W* для Windows, без всякой буквы для DOS, *2* для расширителя Rational DOS или *3* для расширителя PharLap DOS.

Файл PWSOCK.DLL фирмы DTC — это DDL-версия библиотеки большой модели Windows. Две клиентские прикладные программы PATHWORKS V5, которые используют эту DDL, — это протокол передачи файла для Windows (File Transfer Protocol for Windows — FTPW) и "пробный шар" для Windows (Ping for Windows — PINGW). Чтобы использовать PWSOCK.DLL, вы должны осуществить компоновку с PWIMPORT.LIB и малой статической библиотекой под именем *x*PWLINK.LIB, где *x* обозначает модель памяти.

Все гнездовые библиотеки также экспортируют гнездовой интерфейс, не зависящий от языка и модели и уникальный в PATHWORKS, который DEC называет вызовами SKT. Например, совместимый с BSD метод доступа к гнезду DECnet состоит в осуществлении вызова:

```
socket (AF_DECnet, SOCK_STREAM, DNPROTO_NSP);
```

Способ SKT для реализации этого — это осуществить вызов:

```
SktSocket(AF_DECnet, SOCKSTREAM, DNPROTO_NSP, &error);
```

PWIMPORT определяет интерфейс SKT. Файлы *x*PWLINK.LIB определяют традиционный интерфейс BSD, но на самом деле не включают в себя код программы. Они включают вызов в вызовы SKT PWSOCK.DLL. Прикладная программа может осуществлять вызовы SKT, но в этом редко имеется необходимость, и WINSOCK.DLL обычно может обрабатывать эти вызовы.

Если вы хотите обойти Windows Sockets, тем не менее, запустите свои прямые программы гнезд BSD внутри Windows. Вам следует включить PRGPRE.H и осуществить компоновку с библиотекой гнезд, соответствующей вашей модели памяти. Например, если вы выбрали среднюю модель памяти, осуществите компоновку со средней библиотекой гнезд PATHWORKS для Windows (MPWSOCKW.LIB).

Под DOS вы не имеете опции пакетирования *a*PWSOCK.LIB в DLL, т.е. файл .LIB описывает функции, а .DLL реализует код программы. Библиотеки DLL не поддерживаются на непосредственном уровне MS-DOS. В случае написания кода программы на уровне WINDOWS вы имеете опцию пакетирования файла .LIB в DLL. Основным преимуществом этой стратегии кодирования является то, что действительный код программы не встраивается в файл .EXE. Следовательно, если многократные программы, соединенные с .DLL, работают в одно и то же время, то одновременно в физической памяти будет находиться только одна копия используемого совместно кода DLL.

При переносе прямой программы гнезд в языке C, написанной для операционной системы MS-DOS, вы должны решить, собираетесь ли вы также переходить на стандарт WINSOCK. Если да, то вам будет нужно изменить файл включения на *#include* WINSOCK.H.

Версия V5.0A PATHWORKS WINSOCK.DLL поддерживает свой стек версии V5.0A TCP/IP, а также DECnet. Стек TCP/IP, который фирма DEC обеспечивала со своей версией 2.0 клиента TCP/IP, был стеком 3Com. 3Com обеспечила собственный WSOCKETS.DLL, который позволяет этому стеку работать внутри Windows. Этот DLL несовместим со стеком PATHWORKS V5. Эти понятия будут уточнены примерами программирования, которые будут приведены позже.

Использование множественных протоколов: WINSOCK

Следующий факт должен быть известен при работе с любой прикладной программой гнезд Windows, которая использует в настоящий момент только TCP/IP и хочет использовать несколько протоколов — например, DECnet и TCP/IP под продуктом PATHWORKS V5. Программа должна “знать”, что она поддерживает

как DECnet, так и TCP/IP. В этом смысле не имеется возможности сделать так, чтобы WINSOCK.DLL выбросил (fake out) программу клиента.

Например, когда программа создает гнездо, она сообщает файлу WINSOCK.DLL, какой протокол следует использовать например, *AF_DECnet* или *AF_INET*. Сам WINSOCK не может знать, следует ли ему заменять вызовы *AF_INET* вызовами *AF_DECnet*.

Большинство программ, которые используют сеть, используют поиск имени. Для TCP/IP это имя хоста, которое либо находится в файле HOSTS, либо может быть найдено с помощью BIND-совместимого сервиса обработки имен (BIND — Berkeley Internet Name Domain — межсетевой домен имен фирмы Berkeley). Для DECnet — это узловое имя, которое можно найти в DECNODE.DAT или, для V5 PATHWORKS, — в PWNODE.DAT. WINSOCK.DLL не может знать, что именно требует клиентская программа. WINSOCK.DLL обеспечивает функцию *gethostbyname()* для поиска хостовых имен и *getnodebyname()* для поиска узловых имен. Прикладная программа должна определить, какую функцию следует выбрать.

Таким образом, программа не только должна перекомпилироваться и перекомпоновываться в вышеупомянутом случае, но также должна и переписываться для поддержки DECnet в дополнение к TCP/IP.

Выводы относительно Windows Sockets

Основные требования для программирования для WINSOCK с PATHWORKS, а также для любых гнезд TCP/IP Windows:

1. Получить рабочую версию WINSOCK.DLL и поместить ее в переменную DOS PATH.
2. Прогнать выполняемую программу, которая написана для включения *(#include)* файла WINSOCK.H и для использования вызовов модели программирования WINSOCK. Вам следует осуществить компоновку с WINSOCK.LIB. Модель WINSOCK.DLL может быть выбрана в рамках прикладной программы на Visual C++ фирмы Microsoft выбором меню Options, а затем Project. Выберите библиотеку Windows dynamic-link library в диалоговом окне Project Type.

ВЫЗОВЫ УДАЛЕННЫХ ПРОЦЕДУР

Введение

Вызовы удаленных процедур (Remote Procedure Calls — RPC) расширяют механизм вызова процедуры от среды автономного компьютера к распределенной среде, или среде с дистанционным (удаленным) использованием компьютеров. Отметим, что вызов процедуры часто называют вызовом стандартной подпрограммы. Рассмотрим несколько архитектур RPC в среде UNIX, затем исследуем

реализацию RPC фирмы Microsoft с NT. В этой книге примеры программирования RPC не приводятся.

Методология RPC

Вызовы удаленных процедур (RPC) позволяют прикладному программисту писать программу на более высоком уровне абстракции по сравнению с большинством механизмов взаимодействия процессов (IPC). В большинстве случаев программист может игнорировать внутренний механизм связи, который с вызовами RPC становится прозрачным.

В этой среде клиент RPC осуществляет вызовы удаленных процедур для затребования определенных операций. Клиент не знает, как реализован местный интерфейс с "фиктивным модулем" (stub) клиента, и может даже не знать местоположение сервера, выдающего ответы на этот интерфейс. Программа сервера получает запросы от клиентов, выполняет операции на определенных объектах, затем выдает клиенту результаты этих операций.

Объекты — это логически целостные элементы, такие, как база данных, специализированный процессор (процессор программного обеспечения), файл, принтер и т.д. Каждый объект имеет тип — например, очереди на принтер могут быть объектами типа printqueue.

Программы, написанные со стандартными программами RPC, осуществляют доступ к этим объектам через интерфейсы. Интерфейс — это набор определенных операций, которые могут быть применимы к любому из этих объектов.

Вызовы удаленных процедур используют механизм IPC гнезд фирмы Berkeley в качестве своего внутреннего средства сетевой связи. Гнезда — это конечные пункты связи в виде очереди сообщений. Сервер RPC "слушает" одно или несколько гнезд и принимает любой запрос, подающийся на гнездо, которое он "слушает".

Клиенты и серверы RPC связаны — в смысле модулей объектов, объединенных с компоновщиком, — с *фиктивными модулями* (*stubs*). Фиктивные модули — это модули, генерируемые компиляторами из определения интерфейса, написанного пользователем. Когда клиент вызывает местную процедуру, он затрагивает стандартную программу в фиктивном модуле клиента. Фиктивный модуль клиента затем *преобразует* (*unmarshalls*) величины входных параметров в пакет RPC и создает одно или несколько сетевых сообщений для отсылки к серверу. Обычно эти сообщения требуют системного вызова в локальное ядро и передаются либо протоколом на основании логического соединения, таким, как TCP, либо протоколом без логического соединения, таким, как UDP.

Фиктивный модуль сервера ожидает запросов клиента в удаленной системе. Когда он получает запрос по конкретному порту, он преобразует аргументы из сетевых

пакетов, затем выполняет местную процедуру вызова для действительной операции с помощью распакетирования аргументов.

Когда процедура сервера закончила выполнение, она возвращает результаты на фиктивный модуль сервера, возвращая все значения возврата из вызова. После преобразования возвращенных значений фиктивный модуль клиента возвращается к прикладной программе клиента. Это имеет вид обычного возврата от вызова процедуры к прикладной программе клиента, как будто бы это был вызов местной процедуры (стандартной подпрограммы).

На рис. 2-9 проиллюстрирован только что описанный метод RPC.

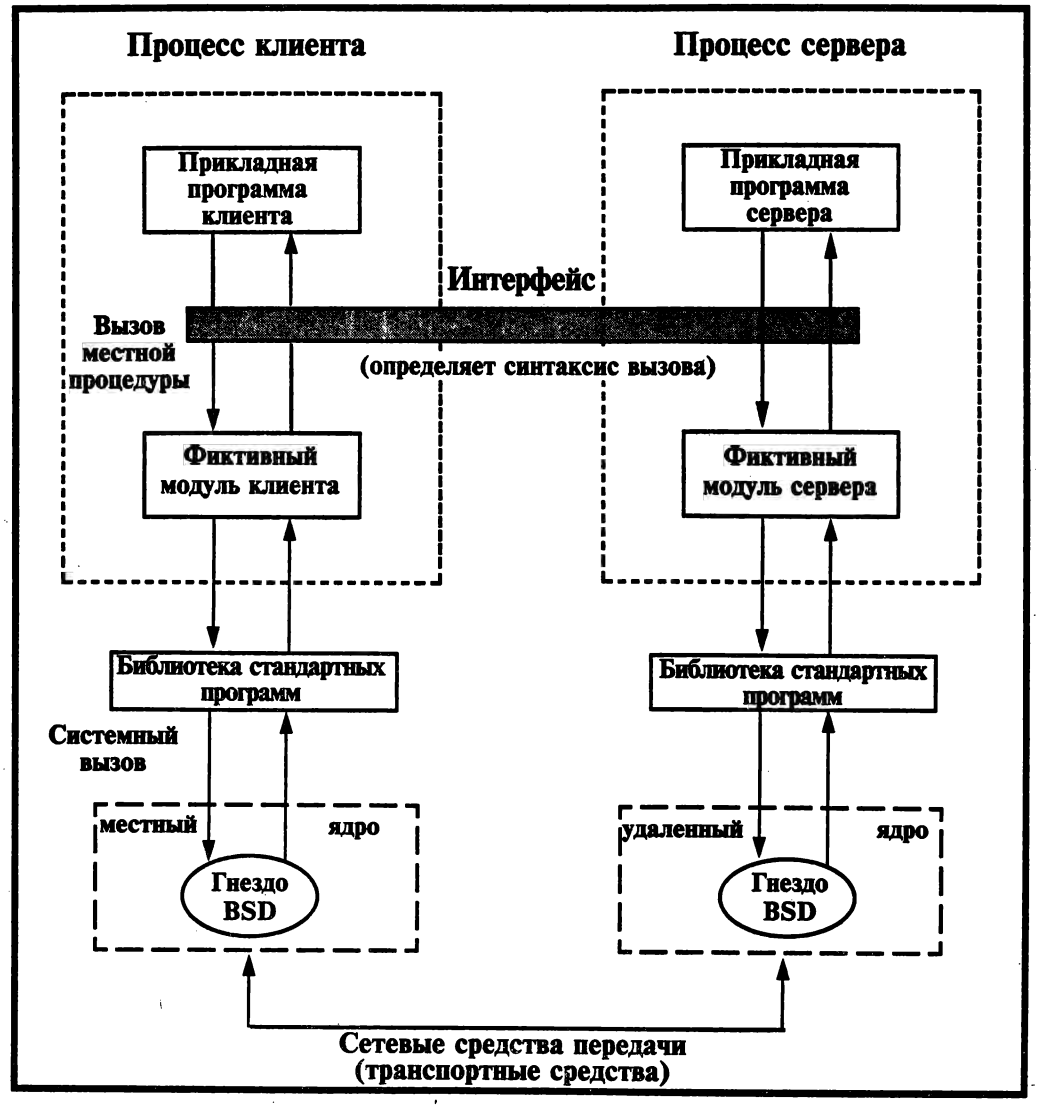


Рис. 2-9. Метод RPC.

Реализации RPC: UNIX

Все реализации RPC имеют сходную архитектуру. Имеются четыре следующих основных реализации RPC:

1. Реализация RPC фирмы Microsoft на системах NT и NTAS, которая имеет четыре компоненты: компилятор языка определений интерфейса Microsoft (Microsoft Interface Definition Language — MIDL), библиотеки, предназначенные для рабочего использования, или рабочие библиотеки, сервис имени и транспортные средства. RPC фирмы Microsoft обеспечивает транспортную поддержку для именованных программных каналов, NetBEUI, NetBIOS, NetWare, использующей протокол SPX NWLink, и TCP/IP.

2. Сетевая вычислительная система (Network Computing System — NCS) Apollo, которую фирма Digital Equipment Corporation реализовала в своих выпусках ULTRIX, например BSD-совместимый UNIX. NCS/RPC Apollo определяет протокол вызова удаленной процедуры и включает в себя три компоненты. Первый компонент, представление сетевых данных (Network Data Representation — NDR), определяет стандарт представления данных. NDR устраняет заботы, связанные с упорядочением байтов и другими архитектурными различиями памяти данных в разных компьютерах. В качестве транспортных средств Apollo использует либо UDP, либо собственный DDS. Двумя другими компонентами NCS Apollo являются компилятор языка определения сетевого интерфейса (Network Interface Definition Language — NIDL), который передает NIDL в фиктивные модули клиента и сервера, и библиотека, предназначенная для рабочего использования (рабочая библиотека).

3. Более ранней реализацией RPC является Xerox Courier. Эта модель построена на транспортных средствах на базе логических соединений протокола упорядоченных пакетов XNS (XNS Sequenced Packet Protocol — SPP) и стандарте представления данных.

4. Система вычисления в открытых сетях (Open Network Computing — ONC) фирмы Sun Microsystem, частью которой являются RPC и стандарт представления внешних данных (eXternal Data Representation — XDR). В этой среде в качестве транспортных средств используется UDP либо TCP. Операционная система OSF/1 фирмы DEC поддерживает ONC RPC.

   Примечание: Более подробную информацию по этой теме смотрите в описании Стивена (Steven) в номерах 2-4 справочника по сетевому программированию UNIX — "*UNIX Network Programming*".

Большая часть терминов, описывающих вызовы удаленных процедур, относится ко всем реализациям RPC. Используемая здесь терминология относится к реализации фирмы Microsoft.

Описатели и привязка

Когда клиент осуществляет вызов удаленной процедуры, запрашивая выполнение операции на том или ином объекте, рабочей библиотеке необходимо иметь информацию для передачи вызова:

- об объекте, на котором должна быть выполнена операция,
- о местоположении сервера RPC, который экспортирует интерфейс, содержащий операцию.

Клиент представляет эту информацию в виде *описателя* (*handle*), который представляет собой указатель структуры данных. Рабочая библиотека обеспечивает несколько стандартных программ для создания и управления таких описателей.

После своего создания описатель всегда представляет один и тот же объект. Однако в разные моменты времени он может представлять различные серверы или же вообще может не представлять никакого сервера. Расположение сервера, представленное в указателе, называется *привязкой* (*binding*). Привязать (*bind*) объект — это значит установить расположение его сервера и установить правильное функционирование сервера на удаленной системе. Этот термин также применяется с *вызовом привязывания* (*bind call*) при программировании гнезд BSD. Фирма Microsoft определяет *привязывающие описатели* (*binding handles*) как описатели в файле прикладной конфигурации (Application Configuration File — ACF), который представляет собой входной файл компилятора MIDL. Привязывающий описатель в действительности является указателем структуры данных, которая представляет собой привязку системы "клиент-сервер" и которая доступна только рабочим библиотекам RPC.

В файле прикладной конфигурации (Application Configuration File — ACF) определены три типа привязывающих описателей:

- implicit_handle (неявный описатель),
- explicit_handle (явный описатель),
- auto_handle (автоматический описатель).

Эти описатели подразделяются на две категории привязывания: *автоматическое* (*automatic*) и *управляемое прикладной программой* (*application managed*). Автоматические описатели привязки используют сервис Locator, или сервис имени, для нахождения сервера и осуществления соединения.

Сервис Locator в RPC

В этом случае сервис *RPC Locator* NTAS аналогичен сервису *устройства отображения портов* (*port tapper*), используемому в реализации RPC фирмы Sun Microsystem. С помощью устройства отображения портов клиент обменивается сообщением с устройством отображения портов для определения адреса сервера.

Когда сервер RPC стартует, он сообщает устройству отображения портов, какой номер порта TCP или UDP он "слушает" и какие программные номера RPC он приготовился обслужить. При правильной установке с помощью сервиса поиска Желтых страниц устройство отображения портов может обрабатывать широковещательные запросы программ/процедур RPC на удаленных хостах и передавать

эту информацию обратно клиенту RPC. Процесс устройства отображения портов стартует как демон (фоновый процесс), когда операционная система загружается.

По методологии сервиса имен фирмы Microsoft клиент, который также использует сервис имен, посылает свой первый запрос на хост сервиса имен. Сервис *Locator RPC* — это имя сервиса, обеспечиваемого NTAS, старт которого осуществляется с интерфейса символов/меню Control Panel/Services (панель управления/виды сервиса).

Сервер имен поддерживает глобальную базу данных сервиса имен RPC, которая включает в себя информацию относительно того, какие виды сервиса какими серверами предлагаются. Сервер имен отвечает клиенту информацией об имени, адресе и протоколах, поддерживаемых сервером, виды сервиса которого затребованы процедурой. Затем клиент "говорит" непосредственно с этим сервисом. Широковещательные механизмы используются, когда Locator не имеет в своей базе данных информации относительно затребованного клиентом сервиса.

Программирование RPC в системе "клиент-сервер"

На рис. 2-10 представлены шаги, осуществляемые при создании программы клиента и сервера в среде RPC Microsoft.

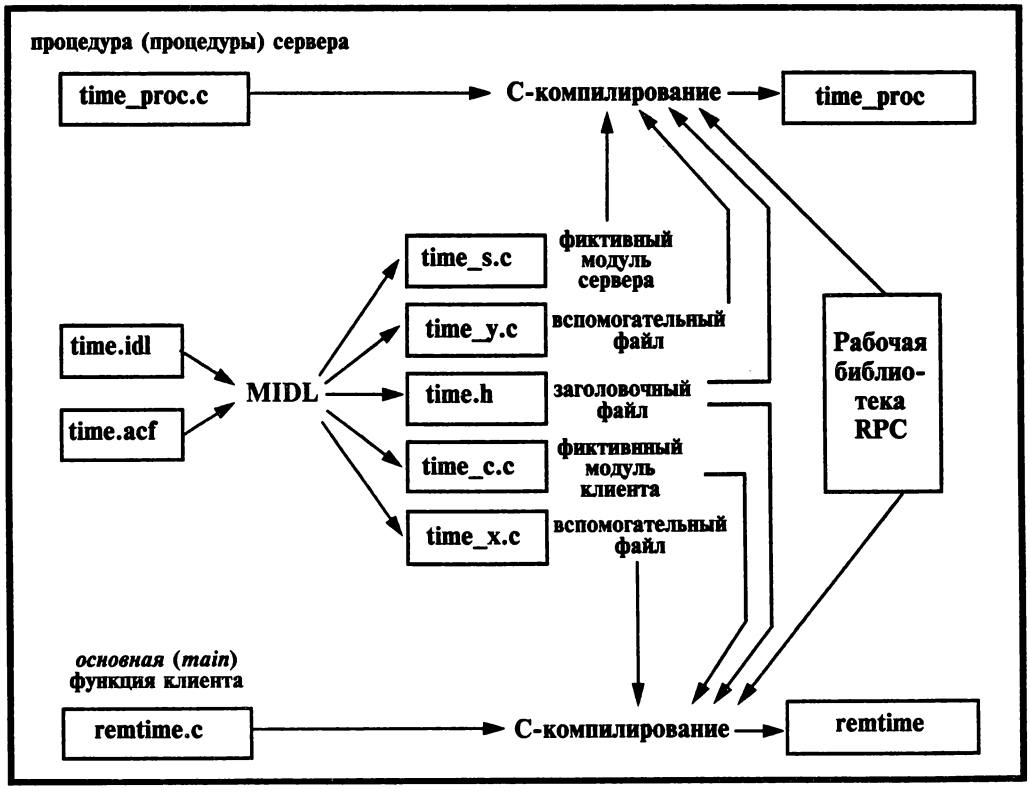


Рис. 2-10. Создание программ клиента и сервера в среде RPC Microsoft:

Как показано на рисунке, компилятор MIDL принимает входные файлы *.idl* и *.acf* и генерирует пять файлов: заголовочный файл, фиктивный модуль и вспомогательные файлы клиента, фиктивный модуль и вспомогательные файлы сервера. *Основная* (*main*) функция клиента должна включать в себя заголовочный файл, генерируемый компилятором MIDL, а также *rpc.h* и/или заголовочный файл *rpcndr.h*. Процедура сервера должна включать в себя заголовок, генерируемый компилятором MIDL.

При использовании RPC имеют место следующие типичные шаги:

1. Программа сервера — в настоящем примере это time_proc — стартует первой. Она может создать гнездо UDP и привязать любой локальный номер порта к гнезду. Затем она осуществляет вызов функции рабочей библиотеки для регистрации себя с местным устройством отображения портов или именем используемого вида сервиса. Эта регистрация сообщает устройству отображения портов информацию о программе и о том, что она делает, например о видах сервиса, предлагаемых в среде RPC. Это осуществляется программой фиктивного модуля сервера.

2. Прикладная программа клиента — remtime — вызывает функцию для уточнения информации об имени сервера, программе в удаленной системе и используемом протоколе. Эта функция входит в контакт с устройством отображения портов или сервисом имен в удаленной системе для отыскания номера порта UDP для сервера.

3. Затем программа клиента вызывает процедуру time_c, которая была определена в фиктивном модуле клиента. Эта процедура была определена как вход на компилятор MIDL. Эта функция посылает дейтаграмму на сервер RPC, используя номер порта из предыдущего шага.

4. Программа сервера получает запрос, определяет, какая местная процедура вызывается, и запускает эту процедуру. Когда вызов этой функции возвращается на фиктивный модуль сервера, фиктивный модуль преобразует эту величину в надлежащий формат и упаковывает ее в соответствующую дейтаграмму для отсылки обратно клиенту.

5. Фиктивный модуль клиента получает дейтаграмму, принимает информацию, преобразует ее, как это требуется, и возвращает ее на программу клиента, осуществившую запрос.

   Примечание: За получением более подробной информации по RPC обращайтесь к соответствующей документации Microsoft, а также см. книгу *UNIX Network Programming* (McGraw-Hill,1990).

ИМЕНОВАННЫЕ ПРОГРАММНЫЕ КАНАЛЫ

Введение

IPC включает в себя сетевые *именованные программные каналы* (*networked named pipes*). Сетевой именованный программный канал — это средство двунаправленной связи IPC, которое разрешает двум удаленным процессам связываться и совместно пользоваться данными в сети.

Сетевые именованные программные каналы являются двунаправленными и в среде BSD UNIX они формируются системным вызовом *пары гнезд (socketpair())*. Заметим, что они также реализуются в среде SystemV. Этот вызов реализуется только в домене UNIX, например, *AF_UNIX*. Этот вызов аналогичен системному вызову *pipe()*, с тем исключением, что *пара гнезд* (*socketpair*) возвращает пару гнездовых дескрипторов, а не файловых дескрипторов. Кроме того, каждый гнездовой дескриптор является двунаправленным, в отличие от программных каналов, которые являются однонаправленными. Эти гнездовые дескрипторы обычно называются *поточными программными каналами* (*stream pipes*), которые и ориентированы на логическое соединение.

От поточных программных каналов в среде "клиент-сервер" были произведены сетевые именованные программные каналы. Сетевые именованные программные каналы поддерживаются в UNIX, OS/2, NTAS и некоторых средах DOS.

Сначала мы исследуем именованные программные каналы — а не сетевые именованные программные каналы — с чисто локальной точки зрения, так как это и есть первоначальное понятие слова "программный канал" (pipe). Именованный программный канал относится к типу "первым вошел — первым вышел" (First In, First Out — FIFO), что означает, что FIFO является однонаправленным потоком данных, т.е. первый байт, записанный в программный канал, является первым байтом, считанным из него.

FIFO подобны стандартным программным каналам UNIX, с тем исключением, что FIFO имеют имя, связанное с ними, которое позволяет независимым процессам иметь доступ к одному именованному программному каналу. Это устанавливает основы для сетевых именованных программных каналов. Процессы на других хостовых системах, которые являются независимыми, могут обмениваться данными через именованные программные каналы.

Программные каналы, созданные системным вызовом программного канала (*pipe()*) в UNIX, могут быть использованы только между процессами, которые порождены одним родителем. Программный канал дублируется от одного процесса к другому системным *вилочным* (*fork()*) вызовом, означающим, что любые дескрипторы открытого файла — следовательно, программные каналы — используются совместно родительским и дочерним процессами после *вилки* (*fork()*).

FIFO создаются системным вызовом *mknod()* в UNIX System V. Они не были доступны в большинстве реализаций V4.3BSD UNIX. Когда создается FIFO, он должен быть открыт как для чтения, так и для записи. Это требует трех отдельных системных вызовов: *mknod()*, *open()* и снова *open()*. Вызов *pipe()* осуществляет то же самое одним вызовом.

Понятие именованных программных каналов

Принцип связи клиента и сервера через именованные программные каналы показан на рис. 2-11.

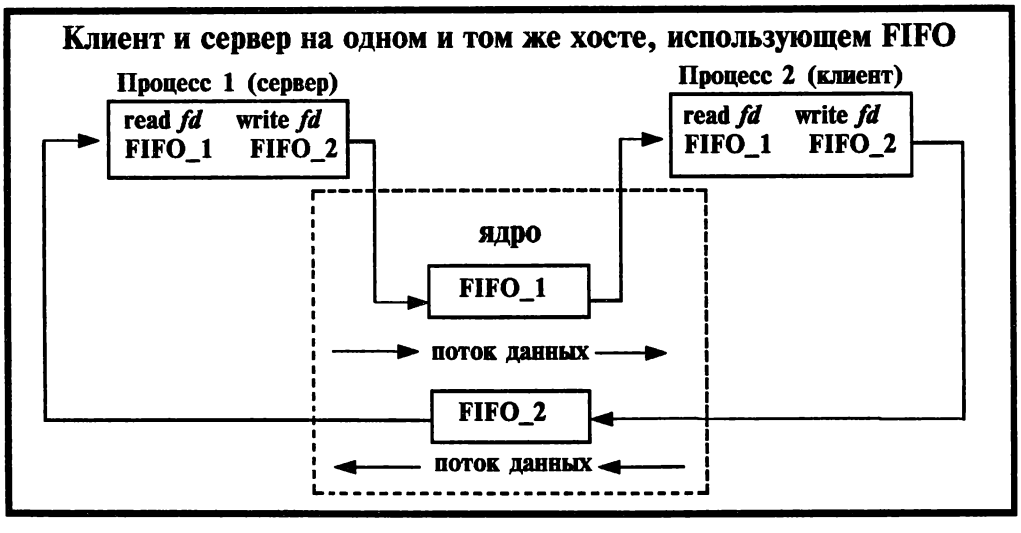


Рис. 2-11. Использование именованных программных каналов для связи в системе "клиент-сервер".

На этой схеме принимается, что процесс сервера создал оба именованных программных канала с помощью *mknod()*.

Вызовы именованных программных каналов

В OSF/1 фирмы DEC *mknod()* принимает следующую форму:

```
int mknod (char *path, int mode, int dev),
```

где *path* — это стандартное имя пути доступа UNIX, включая имя FIFO. Отметим, что FIFO также называется *файлом специального устройства* (*special device file*). Аргумент режима определяет слово режима доступа файла, например чтение и запись для владельца, группы и среды. Слово режима доступа логически обрабатывается схемой ИЛИ (OR) с флагом S_IFIFO из *<sys/stat.h>{usr/sys/include/sys/stat.h}*. Для именованных программных каналов аргумент *dev* игнорируется.

Именованный программный канал может также быть создан командой оболочки *mknod*, размещенной в */usr/sbin/mknod* под OSF/1. Например,

```
sandpiper.alf.dec.com>mknod FIFO_1 p
```

создает именованный файл, называемый FIFO_1, обычно в текущем каталоге. Флаг *p* вызывает создание FIFO. В листинге файла *p* стоит в первой знаковой позиции, которая представляет файловый режим.

```
prw-r_r_ 1 barfield guest 0 Dec 29 12:22 FIFO_1
```

Поточные программные каналы (Stream Pipes) используются в реализациях 4.3BSD и System V UNIX. В реализации System V поточные программные каналы используются между родительским и дочерним процессами на том же самом компьютере, в то время как именованные поточные программные каналы обычно используются независимыми процессами, которые хотят "говорить" друг с другом. Именованные поточные программные каналы обеспечивают основы сетевых именованных, или поточных, программных каналов. Поточные программные каналы являются полностью дуплексными каналами, т.е. двунаправленными.

Сетевые именованные программные каналы под NT

В среде NT и NTAS существует 11 функций программных каналов для манипулирования программными каналами. Некоторые из них отводятся для манипулирования *анонимными программными каналами* (*anonymous pipes*), которые являются однонаправленными программными каналами без имени, и передают данные между родительским и дочерним процессами или между двумя дочерними процессами одного и того же родительского процесса. В наших примерах мы эти вызовы использовать не будем.

Функции именованных программных каналов в среде NT таковы:

- *CallNamedPipe (вызвать именованный программный канал)*
- *ConnectNamedPipe (соединить именованный программный канал)*
- *CreateNamedPipe (создать именованный программный канал)*
- *DisconnectNamedPipe (отсоединить именованный программный канал)*
- *GetNamedPipeHandleState (получить состояние описателя именованного программного канала)*
- *GetNamedPipeInfo (получить информацию именованного программного канала)*
- *PeekNamedPipe (взять из ячейки ОЗУ именованный программный канал)*
- *SetNamedPipeHandleState (установить состояние описателя именованного программного канала)*

- *TransactNamedPipe (обработать именованный программный канал)*
- *WaitNamedPipe (ожидать именованный программный канал)*

Выводы

В этой главе обсуждены три основных принципа программирования IPC: гнезда, вызовы удаленных процедур (Remote Procedure Calls — RPC) и сетевые именованные программные каналы.

- Гнезда TCP ориентированы на логическое соединение, в то время, как гнезда UDP — это гнезда без логического соединения. В среде TCP/IP программы RPC имеют сетевые гнезда в качестве своего внутреннего транспортного механизма.
- Windows Sockets (WINSOCK) является спецификацией Microsoft, которая определяет интерфейс сетевого программирования для Microsoft Windows. Windows Sockets основана на реализации BSD (V4.3) программирования гнезд TCP/IP.
- Вызовы RPC позволяют прикладному программисту писать программу на высоком уровне абстракции по сравнению с большинством механизмов IPC. В большинстве случаев программист может игнорировать внутренний транспортный механизм связи, который становится прозрачным с вызовами RPC. В этой среде клиент RPC осуществляет вызовы дистанционных процедур для затребования выполнения определенных операций. Клиент не знает, как реализован локальный интерфейс для фиктивного модуля клиента и может даже не знать местоположение сервера, передающего ответы на интерфейс. Программа сервера получает запросы от клиента, выполняет операции на различных объектах, затем возвращает результаты этих операций клиенту.
- В определение IPC были включены сетевые именованные программные каналы. Сетевой именованный программный канал — это двунаправленное коммуникационное средство IPC, которое позволяет двум удаленным процессам связываться друг с другом и пользоваться одними данными в сети. Поточные программные каналы являются двунаправленными программными каналами, реализованными под 4.3BSD System V UNIX. Из этой реализации были произведены сетевые именованные программные каналы. Процесс, который создает сетевой именованный программный канал, называется сервером, и клиент устанавливает соединение с именованным программным каналом. Сетевые именованные программные каналы поддерживаются в средах UNIX, OS/2 и NTAS.

Глава 3

Гнезда TCP/IP в системе "клиент-сервер": DOS и BSD UNIX

ВВЕДЕНИЕ

Описание

В этой главе мы рассмотрим гнездовую программу в системе "клиент-сервер", использующую различные стеки TCP/IP у клиента DOS. Позже мы рассмотрим более сложную эхо-программу системы "клиент-сервер" на стеке TCP/IP Windows NT.

Программа DOS в этой главе должна уметь правильно "разговаривать" с тремя различными серверами TCP/IP: ULTRIX (V4.3) фирмы DEC, OSF/1 (V2.0) фирмы DEC и сервером NTAS фирмы Microsoft (V3.1 и выше). У клиента DOS демонстрируемая программа относится к следующим стекам с возможными незначительными модификациями в рамках платформ:

- новый стек TCP/IP ОС PATHWORKS V5 фирмы DEC,
- стек TCP LAN Manager V2.1 фирмы Microsoft, который первоначально был разработан фирмой Hewlett-Packard и 3Com,
- более ранний стек TCP/IP фирмы DEC для ее клиента TCP/IP ОС PATHWORKS V2.0. Этот стек является также стеком фирм Hewlett-Packard и 3Com.

КЛАВИАТУРНОЕ ЭХО В СИСТЕМЕ "КЛИЕНТ-СЕРВЕР"

Гнезда TCP: программы клавиатурного эхо в системе "клиент-сервер"

Этот первый пример системы "клиент-сервер" под названием "клавиатурное эхо" (Keyboard Echo) в системе "клиент-сервер" (см. рис. 3-1), демонстрирует возвращение сервером данных, которые набраны на клавиатуре клиента. Программа использует гнезда BSD. Код программы написан в С.

Первоначальная программа сервера была адаптирована из программы сервера TCP, приведенной в книге Ричарда Стивена (W.Richard Steven) "Программирование

в сетях UNIX" — *UNIX Network Programming* (Prentice Hall, 1990). Эта книга *должна* быть настольной для серьезного разработчика программного обеспечения гнезд.

Эта программа открывает поточное гнездо TCP, привязывает гнездо, устанавливает очередь на прослушивание и принимает любые запросы на соединение, поступающие на гнездо. После установления сеанса связи дочерний процесс оперирует с гнездовым вводом/выводом. Эта программа сервера представляет модель *параллельного сервера* (*concurrent server*), в которой родительский процесс слушает запросы на соединение с помощью *accept()*, затем использует *вилку* (*fork()*) для порождения дочернего подпроцесса обработки задач передачи соединения. В этом примере дочерний процесс отражает все данные, принятые на гнезде, обратно, на персональный компьютер клиента. Функция *readline()* считывает гнездо и ожидает разграничительного *перевода строки* (*linefeed*), до того как данные будут окончательно сброшены обратно.

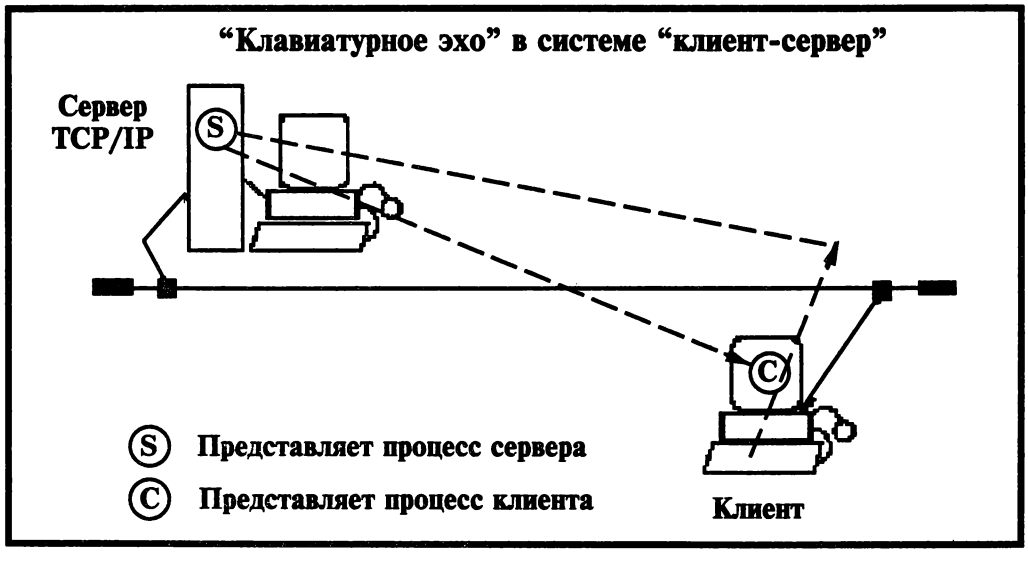


Рис. 3-1. Данные клиента, "отраженные" от сервера.

Программа сервера

```
/*
* FUNCTION to read a stream socket one line at a time, and write each line
* back to the sender.
*
*/
#define    MAXLINE 512

str_echo(sockfd)
```

```
int     sockfd;
{
        int   n;
        char line[MAXLINE];

        for ( ; ; ) {
                n = readline(sockfd, line, MAXLINE);
        if (n ==0)
                return;
        else if (n < 0)
                printf("str_echo: readline error\n");
        if (writen(sockfd, line, n) != n)
                printf("str_echo: writen error\n");
        }
}
/*
*
* FUNCTION: Read a line from a file descriptor. Read the line
* one byte at a time,
* looking for the newline. We store the newline in the buffer,
* then follow it with a null.
* We return the number of characters up to, but not including,
* the null..
*/

int readline(fd, ptr, maxlen)
register      int     fd    /*'register' means value is stored in CPU register*/
register      char    *ptr;
register      int     maxlen;

{
int           n, rc;
char          c;
printf("Enter Readline\n");
    for (n = 1; n < maxlen; n++) {
        if ( (rc = read(fd, &c, 1)) == 1) {

            printf("rec:%d: %c\n", c, c);   /* print decimal value,
                                  * then the character
                                   */
            *ptr++ =c;
            if (c == '\n')
                        break;     /* leave loop now; end-of-line*/
        } else if (rc == 0) {
             if (n == 1)
                        { printf("EOF, no data read\n");
            return(0);            /* EOF, no data read */
            }
                else
                            break;       /* EOF, some data was read */
```

```
            } else
                return(-1);             /* error */
}

    *ptr = 0;
printf("returning from readline(%d)\n",n);
    return(n);
}

/*
*
* FUNCTION: Write 'n' bytes to a (network 'file') descriptor.
* Use in place of the ULTRIX write() call when the
* descriptor is a stream socket.
*
*/

int writen(fd, ptr, nbytes)
register int fd;
register char           *ptr;
register int nbytes;
{

int         nleft,      nwritten;

      printf("writen: (%d) [%s]\n",nbytes,ptr);
      nleft =nbytes;
            while (nleft > 0) {
                        nwritten = write(fd, ptr, nleft);
                        if (nwritten <= 0)
                                    {
                        printf(" Error (%d)\n",nwritten);
                        return(nwritten);       /* error */
                        }
                        nleft -= nwritten;
                        ptr += nwritten;
            }
      printf("string written returning(%d)\n",nbytes-nleft);
            return(nbytes - nleft);
}

/*
* Definitions for TCP server programs.
*/

#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
#include        <arpa/inet.h>
```

```
#include        <varargs.h>
#include        <syslog.h>
#define SERV_TCP_PORT     6000          /*port number, hardcoded */
#define SERV_HOST_ADDR    "16.72.32.196"    /* server address hardcoded*/

char          *pname;

/*
* Example of a server using TCP protocol, with socket programming...
*/

/* #include  </usr/include/arpa/inet.h> */

main(argc,argv)
int           argc;
char           *argv[];
{
            int        sockfd,newsockfd,clilen,childpid;
              struct       sockaddr_in cli_addr,serv_addr;

            pname = argv[0];
            printf("Enter Server Program\n");
            /*
            * Open a TCP socket (an Internet stream socket)...
            */
            if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
            printf("server: cannot open stream socket\n");

/*
* Bind our local address so that the client can send to us...
*/

bzero((char *) &serv_addr,sizeof(serv_addr));
serv_addr.sin_family     = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port       = htons(SERV_TCP_PORT);

if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
            printf("server: cannot bind local address\n");
listen(sockfd, 5);

for ( ; ; ) {
        /*
        * Wait for a connection from a client process.
        * This is an example of a concurrent server, in which the
        * parent forks a child to handle the socket I/O after
        * connection is established...
        */

        clilen = sizeof(cli_addr);
```

```
        printf("starting the accept ...\n");
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        printf("after accept ...\n");
        if (newsockfd < 0)
                printf("server: accept error\n");
        if ((childpid = fork()) < 0)
                printf("server: fork error\n");

        else if (childpid == 0) {                /* child process */
            close(sockfd);
            str_echo(newsockfd);                  /*
                                          * read line from socket,
                                          * then write it back
                                          * to client
                                          */
            exit(0);
        }

        close(newsockfd);                        /* parent process */
        }
}
```

В этом примере программа сервера была компилирована на системе ULTRIX Version 4.3 с помощью следующей команды *cc*:

```
#cc -o tcp_server.exe tcp_srv.c
```

Эта программа полностью совместима с исходной программой ОС OSF/1 фирмы DEC. Та же команда на системе OSF/1 должна перекомпилировать исходную программу без какой бы то ни было модификации. Функция *str_echo()* считывает одну строку из файлового дескриптора, в данном случае из гнезда, и возвращает — пишет через функцию *writen()* — данные обратно клиенту через гнездо. Функция *main()* использует *str_echo()* в следующей структуре:

```
.
.
.
else if (childpid == 0) {                /* child process */
close(sockfd);
str_echo(newsockfd);                      /*
                                  * read line from socket,
                                  * then write it back
                                  * to client
                                  */
exit(0);
```

В этом случае, если работает родительский оператор *fork()*, эта "вилка" возвращает величину 0 дочернему процессу и возвращает идентификатор дочернего процесса родительскому процессу. В противном случае на родительский процесс возвращается величина -1, никакого дочернего процесса не создается, и глобальная

переменная *errno* устанавливается в состояние индикации ошибки. Если дочерний процесс получает величину, равную 0, то первоначальное гнездо закрывается и для эхо-процесса используется новое гнездо, созданное с помощью *accept()*.

Затем дочерний процесс осуществляет вызов *str_echo()*, что использует функцию *readline()* для считывания строки из гнезда, один байт за один раз, в поисках символа новой строки. Новая строка — это символ возврата — восьмиричное число 015, или десятичное 13, — за которым идет подача строки — восьмиричное число 012, или десятичное 10.

Операторы *printf()* были помещены в стратегических пунктах в целях достижения ясности процедур и решения проблем поиска неисправностей. Отметим, что адреса сервера и портов были закодированы жестким кодом в программу, потому что эта программа предназначалась для простой демонстрации работы гнезд. Теперь рассмотрим программу клиента.

Программа клиента

Программа клиента была компилирована с помощью клиентских библиотек TCP/IP PATHWORKS Version 2.0. Стек TCP, использованный с ней, является стеком фирмы 3Com. В принципе, это такой же стек, как и стек DOA V2.1 LAN Manager фирмы Microsoft, который является стеком 3Com/Hewlett-Packard и который фирма Microsoft объединила с LAN Manager для DOS. Существуют незначительные отличия между стеком TCP V2.0 фирмы DEC и стеком V2.2 LAN Manager фирмы Microsoft. У вас не будет проблем при адаптации этой показанной первоначальной клиентской программы к стеку 3Com LAN Manager фирмы Microsoft.

Эта программа компоновалась в соответствии с DOS_SOCK.LIB, который идет в субкаталоге \TCPIP\TOOLS\DOS на дискетах с версией 1.1 и субкаталоге \TCPIP\SOCKET\DOS на второй дискете (из набора из двух дискет) с утилитами TCP/IP версии 2.0 фирмы DEC.

Эта клиентская программа весьма проста. Она открывает гнездо, затем соединяет его с адресом сервера. Когда соединение установлено, клиент набирает строку на клавиатуре и посылает эти данные на сервер.

Затем клиент использует вызов *ioctl()* для определения входящих на гнездо данных и использует *recv()* для захватывания данных. Как и в программе сервера, операторы *printf()* помещаются в стратегические пункты в программе клиента для облегчения использования и для доказательства того, что данные были отосланы/приняты правильно. Эта клиентская программа была компилирована с помощью компилятора C фирмы Borland.

Клиентская программа

```
/*
* Example of a client using TCP/IP protocol stream sockets
* Include files have been given complete paths to show where
* default directories are upon installation of the PC software.
*/

  /* Version 1.1 includes... */

#include    <\tcpip\tools\include\sockdefs.h>
#include    <\tcpip\tools\include\sock_err.h>
#include    <\tcpip\tools\include\netdb.h>
#include    <stdio.h>
#include    <errno.h>
#include    <\tcpip\tools\include\sys\socket.h>
#include    <\tcpip\tools\include\netinet\in.h>
#include    <varargs.h>


/* Version 2.0 includes ... Same filenames; sockets subdirectory
takes place of tools subdirectory * on install floppies
*
*#include    <\tcpip\sockets\include\sockdefs.h>
*#include    <\tcpip\sockets\include\sock_err.h>
*#include    <\tcpip\sockets\include\netdb.h>
*#include    <stdio.h>
*#include    <errno.h>
*#include    <\tcpip\sockets\include\sys\socket.h>
*#include    <\tcpip\sockets\include\netinet\in.h>
*#include    <varargs.h>
*
*/

      /*
      * Server's address and port number are hardcoded into program,
      * since this program only attempts to demonstrate
      * socket-to-socket communications. The server program chose
      * arbitrary port number 6000, since this is not reserved under
      * ULTRIX...
      *
      */

#define SERV_TCP_PORT       6000
#define SERV_HOST_ADDR      "16.72.32.196"

char *pname;

main()
{
  char    buf[128],c;
```

```
int     sockfd;
long    retcode;
long    far *argp;
struct  sockaddr_in  serv_addr;
struct  sockaddr *from;
/*
*
* Fill in the 'serv_addr' structure with the address
* and port number (hardcoded) of the
* server with which we want to "talk" ...
*/

serv_addr.sin_family     = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port       = htons(SERV_TCP_PORT);

/*
* Open a TCP stream socket ...This call on the PC
 * is the same as a UNIX BSD call; with this version of
 * the 3Com 3+Open software, we are limited to SOCK_STREAM
* and SOCK_DGRAM for the second argument ...
*/

if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) < 0)
     {
       printf("client: cannot open TCP stream socket");
       return;
     }
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
     {
       printf("client: cannot connect to TCP server");
       return;
     }
/*
* Server needs a newline character attached to end of
* buffer received; thus, '\n' is placed in buf[len]
* MSG_PUSH parameter on send() call pushes data out
* immediately ...
*/

for (;;)
{
     int       len;
     int       bytesrec;
     printf("Sending a new msg - type 'q' to quit:\n ");
     getline(buf, "Enter msg : ");
     if (buf[0] == 'q') return;
     len = strlen(buf);
     printf("[%d](%s)\n",len,buf);
     buf[len]='\n';
```

```
        buf[len+1]=' .0';
  if(send(sockfd, buf, len+1, MSG_PUSH) != (len+1))
        {
         printf("write socket error(%d)\n",errno);
         close_socket(sockfd);
        return;
        }

         printf("Press any key to read...(q to quit)\n");
        c=getch();
        if ('q'==tolower(c))
          {
                        close_socket(sockfd);
                        return;
              }
        len = 128;
        argp = & retcode;
        if(ioctl(sockfd,FIONREAD,argp))
          {
                  printf(" ioctl failed(%d)\n",errno);
                  close_socket(sockfd);
                  return;
            }
             printf("ioctl says %ld bytes waiting\n",*argp);
    if((bytesrec = recv(sockfd, buf, len, 0 )) < 0)
            {
                  printf("read socket error %d\n",errno);
                  close_socket(sockfd);
                  return;
            }
    printf("recv'd: (%s)\n",buf);

    }
  }
  getline(line, prompt)
  char  *line;
  char  *prompt;
  {
    char    ch;

    printf("%s", prompt);
    while ((ch = getchar()) != '\n' && ch != EOF)
              *line++ = ch;
    *line = ' .';
    if (ch == EOF)
          return(EOF);
  }
```

Функция *inet_addr()* преобразует межсетевой номер в виде последовательности знаков в 32-битовый межсетевой номер, который должен быть послан вместе

с пакетом. Эта функция межсетевой поддержки гнезда имеется также в комплекте средств разработчика программного обеспечения гнезд, версия V5 фирмы DEC. Это стандартная функция поддержки BSD UNIX.

Функция *htons()* превращает 16-битовый номер порта из хостового байтового порядка в сетевой байтовый порядок. Протокол TCP требует, чтобы его номера портов передавались как большой эндиан; в компьютерах Intel эта работа выполняется оператором *htons()*. Этот номер помещается в структуру адресов гнезд в виде короткого (16 бит) целого числа без знака в сетевом байтовом порядке. Затем клиент открывает поточное гнездо TCP и осуществляет соединение со следующей кодовой структурой.

```
if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0)
  {
  printf("client: cannot connect to TCP server");
  return;
  }
```

Отметим, что оператор адреса подается на переменную *serv_addr*. Этот адрес указывает на межсетевую структуру адресов гнезд *sock_addr_in*. Функция *sizeof(serv_addr)* определяет длину структуры имени получателя или структуру *sockaddr_in*.

Входной код программы с клавиатуры клиента является прямым. Он вызывает свою собственную функцию *getline()*, которая использует стандартную функцию ввода/вывода *getchar()* для ввода каждого знака, набранного на клавиатуре. Информация до символа новой строки (включая его), набранного для окончания входной строки, помещается в буфер *buf[]*. После знака новой строки в буфере помещается пустой символ (0).

Затем используется оператор *send()* для отсылки данных на сервер с использованием файлового дескриптора гнезд, полученного с вызовом *socket()*. Оператор *send()* требует адреса буфера, его длину, включая пустой символ, и параметр *flags*.

В этом случае используется флаг MSG_PUSH, который немедленно сбрасывает буфер. Для того чтобы получить данные, "отраженные" от сервера в режиме "эхо", клиент вводит программу приема с помощью *recv()*. Однако сначала он использует вызов *ioctl()* для определения количества байтов, которые ждут приема. Вызов *ioctl()* осуществляет это с помощью параметра запроса FIONREAD, для которого *argp* содержит адрес, который заполняется величиной количества байтов (коротким целым числом), ожидающих приема на гнезде.

Если этот вызов не проходит, гнездо закрывается и управление возвращается из *main()*. В противном случае данные гнезда считываются с помощью *recv()* и печатаются с помощью *printf()*.

Осуществление переноса к клиенту PATHWORKS V5.x

При перенесении этой программы гнезд TCP версии 2.0 фирмы DEC на программу TCP в операционной системе PATHWORKS V5.0A или выше мы использовали программное обеспечение Microsoft Visual C++ Professional Edition Version 1.0 для Windows 3.1 и DOS. Отметим, что версия 2.0 была последней версией к моменту написания этой книги. Visual C++ (V1.0) фирмы Microsoft использует оптимизирующий компилятор V6.0 этой же фирмы.

Дадим следующие советы, которые полезны при перенесении этой программы в файл DOS.EXE для PATHWORKS V5 с использованием программы на Visual C++ фирмы Microsoft:

- Сначала вы должны выбрать соответствующую модель программирования в Visual C++. Мы использовали *MS-DOS Executable (.EXE)*. Она была выбрана в рамках интерфейса меню Options/Project. На рис. 3-2 показаны экраны монитора.
- Вы должны выбрать соответствующую модель памяти в рамках Visual C++. Мы выбрали *среднюю* модель памяти, хотя *малая* модель памяти также хорошо бы работала в этом случае.
- В средней модели памяти DOS.EXE без динамически компонуемой библиотеки для компоновки объекта вы должны создать свою собственную библиотеку объектов, с помощью которой необходимо осуществлять компоновку. Фирма Microsoft предоставляет утилиту *NMAKE.EXE* (в \MSVC\BIN) и формирующий файл — MFCDLL.MAK в \MSVC\MFC\SRC — что позволяет вам построить библиотеку (версия 1.0). Мы использовали следующую команду *nmake*:

```
nmake target+r model+m dll+0 codeview=0 debug=0 /f mfcdll.mak
```

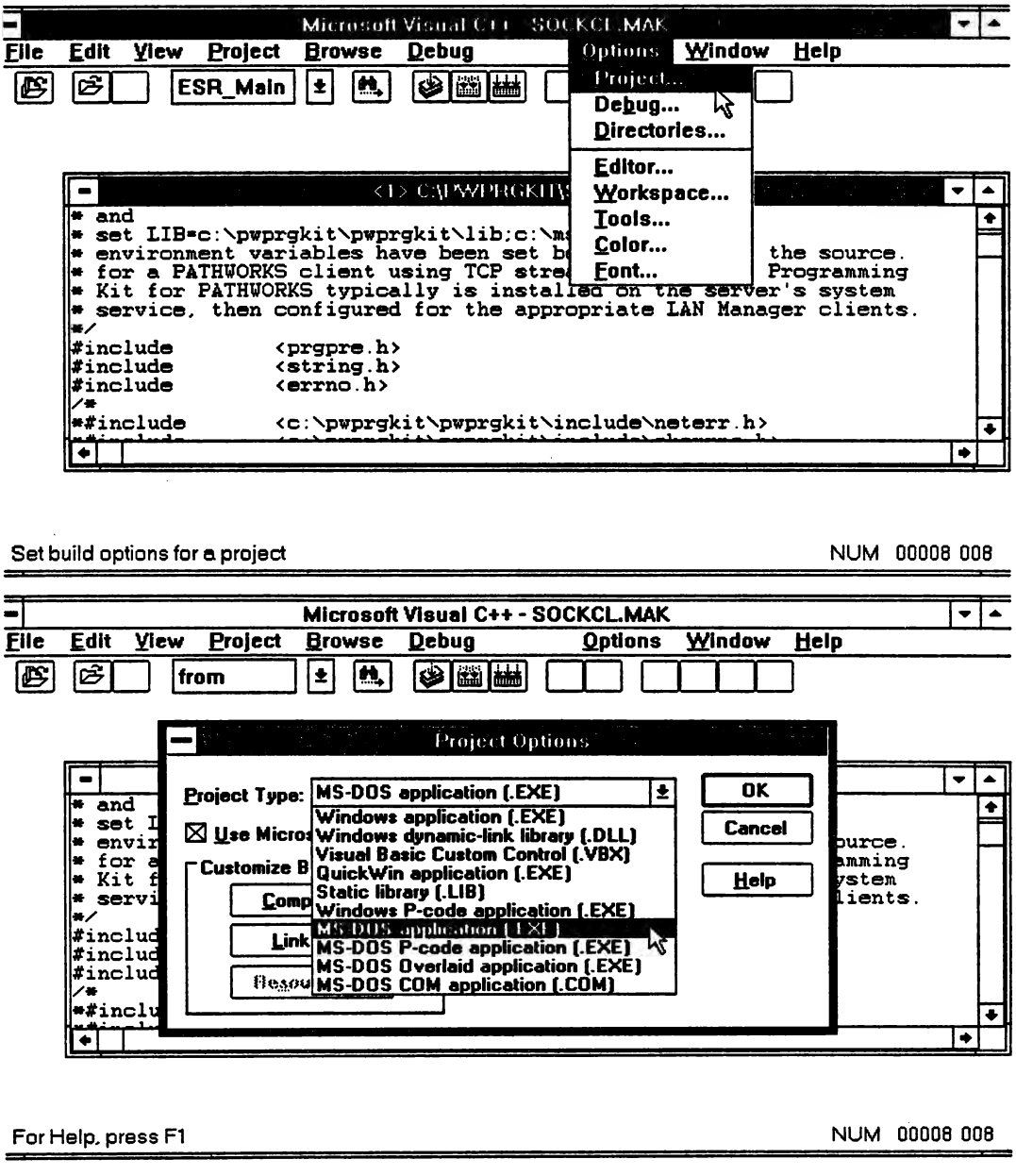


Рис. 3-2. Экраны при переносе к клиенту PATHWORKS V5.x.

Отметим, что фирма Microsoft имеет файл README.TXT в директории \MSVC\MFC\SRC с версией V1.0, что объясняет опции NMAKE при создании соответствующих библиотек объектов. Фирма Microsoft также предоставляет технические замечания с Visual C++, объясняющие этот процесс, которые, возможно, изменились в более поздних версиях.

Это создало библиотеку файлов под названием *mafxcwd.lib*, которая является по умолчанию файловым именем библиотеки с NMAKE. Мы переименовали этот файл в *mafxcr.lib*, вычеркнув W для отражения того факта, что эта модель не является выполнимой моделью Windows, и вычеркнув *d*, чтобы показать, что она не была компилирована для DEBUG. Символ *m* определяет среднюю модель памяти. Часть имени *afx* используется всегда. При определении компонующих библиотек с интерфейсом меню Options/Project вы осуществляете щелчок мышью по клавише Linker внутри первоначального окна, затем выбираете Input — это осуществляется автоматически — из диалогового окна. Наберите на клавиатуре вышеупомянутое имя в соответствующем диалоговом окне, как это показано на рис.3-3.

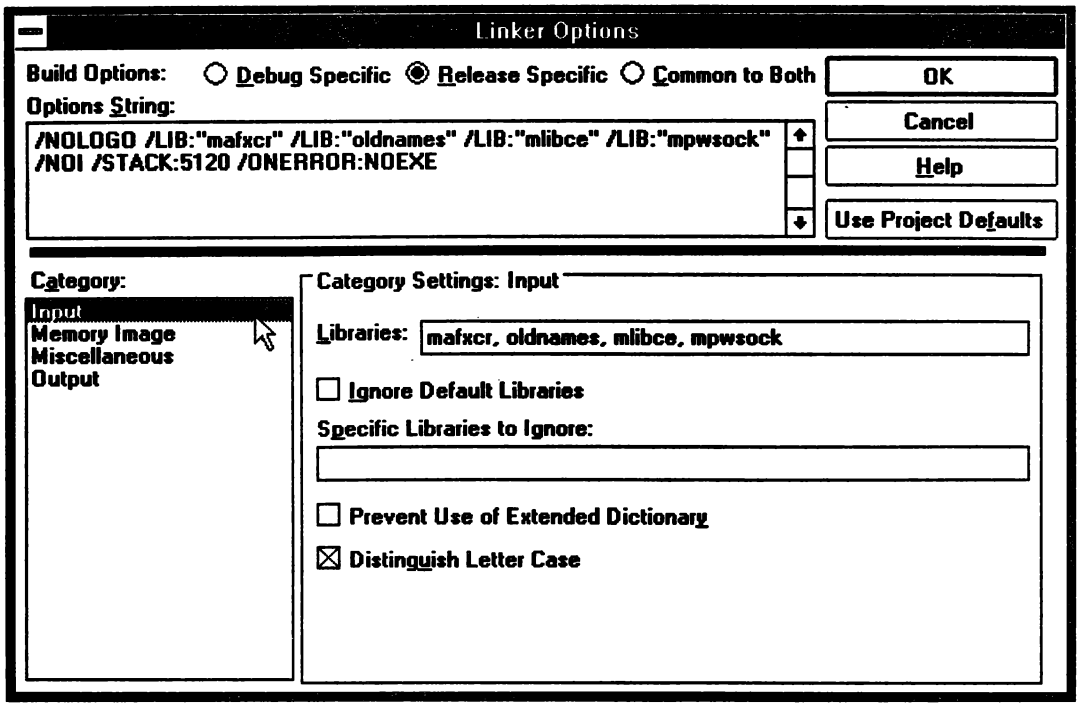


Рис.3-3. Определение компонующих библиотек

- PATHWORKS V5.0A включает в себя заголовочный файл под названием *prgpre.h*, который имеет операторы *#include*, объединяющие большинство заголовочных файлов, необходимых для компилирования, включая файл *stdio.h*. Используйте этот файл с параметром компилятора */Yu" (filename")*, который приказывает "включить версию предварительно скомпилированного заголовка {.PCH} {предварительно скомпилированный заголовок} того же самого файлового имени заголовка, используемого в операторе *#include*." Если используется параметр */Yu"prgpre.h"*, то компилятор знает, что нужно захватить *prgpre.pch* и использовать его, экономя время на компилирование

в последовательных компиляциях, в которых заголовок не меняется. Ниже приведена перенесенная версия текста программы 3Com, показывающая различия в коде программы и вызовах гнезд для клиента TCP/IP PATHWORKS V5.

Перенесенная версия кода программы для Visual C++ с использованием модели C

```
/*
* Example of a client using TCP/IP protocol stream sockets
* Include files assume:
* set INCLUDE=C:\PWPRGKIT\INCLUDE;c:\msvc\include
* and
* set LIB=c:\pwprgkit\lib;c:\msvc\lib
* environment variables have been set before compiling the source.
* The SDK for PATHWORKS typically is installed on the server's system
* service, then configured for the appropriate LAN Manager clients.
*/
#include          <prgpre.h>
#include          <string.h>
#include          <errno.h>

/* Hardcoded includes are as follows:
*
*#include        <c:\pwprgkit\include\neterr.h>
*#include        <c:\pwprgkit\include\skerrno.h>
*#include        <c:\pwprgkit\include\netdb.h>
*#include        <stdio.h>
*#include        <c:\pwprgkit\include\socket.h>
*#include        <c:\pwprgkit\include\netinet\in.h>
*#include        <c:\pwprgkit\include\in.h>
*#include        <varargs.h>
*#include        <c:\pwprgkit\include\sockdefs.h>
*
*/

/*
* Server's address and port number are hardcoded into program,
* since this program only attempts to demonstrate
* socket-to-socket communications. The server program chose
* arbitrary port number 6000, since this is not reserved under
* ULTRIX or OSF/1.
*/

#define   SERV_TCP_PORT     6000
#define   SERV_HOST_ADDR     "16.72.32.195"

char       *pname;

/* function prototype */

getline (char *line, char *prompt);
```

```
void main()
{
        char    buf[128],c;
        int     sockfd;
        long    retcode;
        long      far  *argp;      /* used in sioctl call */
        struct sockaddr_in serv_addr;
        struct sockaddr *from;


        /*
        *
        * Fill in the 'serv_addr' structure with the address
        * and port number (hardcoded) of the
        * server with which we want to "talk" ...
        */

        serv_addr.sin_family        = AF_INET;
        serv_addr.sin_addr.s_addr  =inet_addr(SERV_HOST_ADDR);
        serv_addr.sin_port          =htons(SERV_TCP_PORT);

        /*
        * Open a TCP stream socket ...This call on the PC end
        * is the same as any bsd UNIX call; with earlier version of
        * PATHWORKS TCP client using the 3Com 3+Open stack,
        * we are limited to SOCK_STREAM and SOCK_DGRAM
        * for the second argument ...
        */

        if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) < 0)
              {
                printf("client: cannot open TCP stream socket");
              return;
              }
        if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0) \
              {
              printf("client: cannot connect to TCP server");
              return;
              }

        /*
        * Server needs a newline character attached to end of
        * buffer received; thus, '\n' is placed in buf[len].
        * MSG_OOB (Out-Of-Band) parameter on send() call pushes
        * data out immediately ...
        */

         for (;;)
        {
              int       len;
```

```
                int     bytesrec;
                printf("Sending a new msg - type 'q' to quit:\n ");
                getline(buf, "Enter msg : ");
                if (buf[0] == 'q') return;
                len = strlen(buf);
                printf("[%d](%s)\n",len,buf);
                buf[len]='\n';
                buf[len+1]='/000';

        if(send(sockfd, buf, len+1, MSG_OOB) != (len+1))
        {
                printf("write socket error(%d)\n",errno);
                sclose(sockfd);
                return;
        }

    printf("Press any key to read...(q to quit)\n");
        c=getchar();
        if ('q'==tolower(c))
                {
                sclose(sockfd);
                return;
}

        len = 128;
        retcode = 0;
        argp = &retcode;

        if(sioctl(sockfd,FIONREAD,(unsigned char *)argp))
            {
                printf("sioctl failed(%d)\n",errno);
            sclose(sockfd);
                return;
                }
        printf("sioctl says %ld bytes waiting\n",*argp);
        if ((bytesrec = recv(sockfd, buf, len, 0)) < 0)
                {
                printf("read socket error %d\n",errno);
                sclose(sockfd);
            return;
            }
        printf("recv'd: (%s)\n",buf);
    }
}
getline(char *line, char *prompt)
/*
*char     *line;
*char     *prompt;
*/
```

```
{
        char      ch;
          printf("%s", prompt);
          while ((ch = getchar()) != '\n' && ch != EOF)
                    *line++ = ch;
        *line = '/0';
        if (ch == EOF)
                    return(EOF);
}
```

Различия в перенесенном коде программы по сравнению с V2.0 TCP/IP

Помимо различий в файле включения, первое заметное различие состоит в том, что мы использовали прототип функции для *getline()* до программы *main()* для удовлетворения требований к условным обозначениям C++. Этот прототип был таким:

*getline (char *line,char *prompt);*

Другие заметные отличия (показаны жирным шрифтом) проявляются в следующем коде программы в V5.0A (Примечание: V5.1 была версией клиента, когда писалась эта книга):

```
if(send(sockfd, buf, len+1, MSG_OOB) != (len+1))
 {
  printf("write socket error(%d)\n",errno);
  sclose(sockfd);
  return;
 }
```

Отметим, что флаг *MSG_OOB* используется с вызовом *send()* в PATHWORKS V5.0A в отличие от параметра MSG_PUSH в программе V2.0. Для передачи внедиапазонных (Out-Of-Band) или *срочных* (*expedited*) данных используется MSG_OOB. Это означает, что данные сообщения посылаются немедленно в качестве приоритетного сообщения. MSG_PUSH в программе 3Com V2.0 в основном означает то же самое.

Функция *sclose()* PATHWORKS V5 была *close_socket()* в версии V2.0. Изменилось только имя; функции осуществляются так же. Функция *sclose()* завершает любое неподтвержденное соединение на гнезде, к которому обращается *sockfd*, но этот оператор не затрагивает гнезд с опционным набором *SO_KEEPALIVE*. Эта опция устанавливается, когда вы используете вызов *setsockopt()* с V5. Этот вызов, который не используется в этой программе, принимает следующий вид:

```
#include      <socket.h>
 .
 .
 .
```

```
int setsockopt (sockfd,level,optname,optval,optlen)
int sockfd,level,optname,optlen;
unsigned char *optval;
```

Вызов *optname* определяет идентификатор, такой, как SO_KEEPALIVE, для которого следует установить опцию *setsockopt()*. Этот идентификатор поддерживает гнездо в активном состоянии. Любые каналы и гнезда, связанные с гнездом, остаются активными, несмотря на любые попытки их прекращения, отключения или разъединения. Опции существуют на нескольких уровнях, и вы должны определить уровень для желаемой опции. Например, TCP/IP не имеет каких-либо опций управления доступом, но имеет опции на гнездовом уровне, такие, как SO_KEEPALIVE. Другой константой гнездовой опции является SO_LINGER, которая заставляет вызов *sclose()* задержаться, если имеются данные. Это оставляет непосланные сообщения нетронутыми до тех пор, пока неподтвержденные сообщения не будут переданы.

Вызов *level* определяет уровень, на котором осуществляется манипулирование гнездовыми опциями. Примером является PROTO_TCP для TCP/IP в DOS. Это опция, специфичная для протокола. SOL_SOCKET используется для семейства гнездовых опций, т.е. специфичных безотносительно протокола.

Вызов *optval* определяет адрес буфера, который содержит информацию о гнезде, а *optlen* представляет собой длину буфера.

Другое различие существует в вызове управления вводом/выводом, *sioctl()*, у клиента V5.0A DOS и *ioctl()* у клиента 3Com/Microsoft и DEC V2.0. Вызов V5.0A выглядит следующим образом:

```
if(sioctl(sockfd,FIONREAD, (unsigned char *)argp))
        {
        printf(" sioctl failed(%d)\n",errno);
        sclose(sockfd);
        return;
        }

printf("sioctl says %ld bytes waiting\n",*argp);
        .
        .
        .
```

У клиента V5.0A этот вызов требует, чтобы третий аргумент, *argp*, или указатель аргумента, содержал данные, считываемые вызовом *sioctl()*. FIONREAD возвращает данные, запоминая количество байтов данных, ожидающих на гнезде, в виде указателя на буфер. Тип данных представляет собой *unsigned char *argp*, или указатель на символьные данные без знака.

В версии 2.0 вызов *ioctl()* этот аргумент определяется как *long far *argp*, или дальний указатель на длинное слово. Таким образом, переменная argp, определенная как *long far *argp*, заносится в тип (unsigned char*).

После того как мы рассмотрели эти вопросы переноса, мы успешно скомпилировали код программы и осуществили компоновку с выполнимой в DOS моделью в Visual C++, C. Компонующие библиотеки показаны на предыдущих рисунках.

Глава 4

Клиент и сервер в режиме "эхо": WINSOCK и Windows NT

ВВЕДЕНИЕ

Назначение

Основной целью этого примера является демонстрация простой реализации программирования гнезд со стороны клиента и со стороны сервера. В этом примере создана консольная прикладная программа, использующая стандартные функции гнезд, и обеспечен стандартный набор команд, подлежащих выполнению. Примеры программ клиента и сервера были написаны, компилированы и компонованы в системе Windows NT, однако они должны работать — с минимальными регулировками — в системе Windows V3.1/DOS. В стратегических пунктах рассматриваются проблемы переноса. Примеры программ следуют за объяснениями особенностей модели представления.

Клиент

Программа клиента требует двух аргументов (параметров) командной строки и принимает третий.

- *inet-address* — Этот аргумент представляет межсетевой адрес сервера, выраженный в виде 4-байтовой последовательности. Примером может быть последовательность 16.72.32.185, представляющая собой адрес TCP/IP класса A.
- *port* — Этот аргумент представляет адрес порта, к которому привязывается сервер на конкретной вычислительной машине, на которой реализуется программа сервера.
- *[log-flag]* — Этот дополнительный (опциональный) параметр идентифицирует первоначальное состояние глобальной переменной *LogFlag*. Значение этой переменной, равное 1, означает показ деталей регистрации (log details). Если она не определена, то значение по умолчанию равно 0, т.е. регистрация не осуществляется.

Программа клиента осуществляет следующие шаги:

1. Инициализирует программу.
2. Инициализирует сеть.
3. Создает оконечный пункт.
4. Осуществляет соединение с другим оконечным пунктом (установленным сервером).
5. Осуществляет подсказку в отношении местных данных от пользователя.
6. Посылает данные на сервер.
7. Обрабатывает любые команды в местных (локальных) данных, как это показано ниже:

exit — Local: client! Exit the client; Remote: server! Client is exiting
Местная: клиент! Осуществляет выход клиента; Удаленная: сервер! Клиент выходит из работы. Соответствует нормальному выходу клиента из работы;

quit — Local: client! Exit the client; Remote:server! Exit the server
Местная:клиент!Осуществляет выход клиента; Удаленная: сервер! Осуществляет выход сервера из работы. Соответствует нормальному окончанию работы;

bye — Local: client! Exit the client; Remote: no bye command corresponds to abnormal client exit
Местная: клиент! Осуществляет выход клиента; Удаленная: аварийному выходу клиента из работы не ставится в соответствие никакой команды bye;

clog — Local: client! Toggle the client Log flag
Местная: клиент! — Перебрасывает флаг клиента Log flag;

8. Принимает данные от сервера.
9. Возвращается к шагу 5 для ввода новых данных от местного пользователя.

Сервер

Программа сервера требует одного аргумента командной строки и принимает еще два дополнительных (опциональных) параметра:

- *Port-number* — Это номер порта, отведенный для сервера. Если вы даете значение, равное 0, тогда функция *bind* выберет для вас неиспользованный порт.

- *[wait-flag]* — Этот опциональный аргумент используется для установки первоначального значения переменной *WaitFlag*. Если вы не даете этот аргумент, тогда будет использовано значение по умолчанию — 0. Если значение аргумента *wait flag* равно 1, тогда вам на сервере будет выдана подсказка нажать <Return>, до того как данные будут получены от клиента. Это позволяет вам моделировать условие занятости сервера, когда клиент пытается осуществить отсылку данных.

- *[log-flag]* — Этот опциональный аргумент используется для установления первоначального значения переменной LogFlag. Если вы не даете этот аргумент, то будет использоваться значение по умолчанию, равное 0. Значение, равное 1, означает показ деталей регистрации (log details).

Сервер осуществляет следующие шаги:

1. Инициализирует программу.
2. Инициализирует сеть.
3. Создает оконечный пункт.
4. Регистрирует оконечный пункт в сети.
5. Ожидает соединения с клиентом.
6. Принимает данные от клиента:

   если *[WaitFlag]*, то ожидает, когда пользователь нажмет <Return>; принимает данные.

7. Проверяет сетевые ошибки:

   Connection Reset (сброс соединения);
   Zero Length Data (данные нулевой длины).

8. Обрабатывает удаленные [Remote] команды, посланные от клиента:

   Exit — перейти к Шагу 5, ожидать другого клиента;
   Quit — вывести сервер из работы;
   Slog — перебросить флаг log flag (регистрация) сервера;
   Swait — перебросить флаг wait flag (ожидание) сервера.

9. Модифицирует данные, полученные от клиента.
10. Отсылает данные обратно к клиенту:

    Send the data (отослать данные);
    Check for Network errors (проверить на наличие сетевой ошибки).

11. Переходит к Шагу 6 (прием новых данных от клиента).

КЛИЕНТ В РЕЖИМЕ "ЭХО" (С ПОЯСНЕНИЯМИ)

```
#include <winsock.h>
#include <stdio.h>
#include <stdlib.h>
#include "UserLib.h"

char * NotMe = "Server";
char * Me    = "Client";

int LogFlag;
```

WINSOCK.H

Файл включения winsock.h является необходимым в этой клиентской программе, потому что она создается как консольная программа Windows NT. Для того чтобы изменить ее у клиента MS-DOS, вы должны заменить WINSOCK.H на PRGPRE.H. Помните, что PRGPRE.H подходит для ОС PATHWORKS V5.0A и выше. Если у вас гнездовая библиотека другого поставщика, тогда вы должны будете осуществить включение соответствующего файла этого поставщика.

STDIO.H, STDLIB.H

Эти файлы включаются для предоставления доступа к стандартным функциям библиотеки С, используемым в этой программе для ввода и вывода.

UserLib.h

Этот файл представляет собой специально созданную библиотеку, которая используется в представленных здесь примерах для "завертывания" стандартных вызовов гнезд и переадресации их к функциям *User_x*. Функции *User_x* имеют встроенную логику для вывода на печать различных сообщений при условии, что установлен флаг *LogFlag*.

Глобальные переменные

В файле включения UserLib для идентификации отсылки сообщений клиентом или сервером используются две глобальные переменные *Me* и *NotMe*. Так как клиент и сервер оба вызывают одни и те же функции UserLib, был создан метод, позволяющий определить, кто именно посылает и получает сообщения. В функции отсылки UserLib логика выведет на печать следующее сообщение при условии задействования флага *LogFlag*:

```
The [Me] has sent data to the [NotMe]
printf("The %s has sent data to the %s\n",Me,NotMe)

([Me] отослал данные к [NotMe]
printf("%s отослал данные к %s\n",Me,NotMe).
```

Когда клиент вызывает функцию отсылки UserLib, это сообщение выведется на печать таким образом:

```
"The Client has sent data to the Server"
("Клиент отослал данные на Сервер")
```

Однако, когда сервер вызывает ту же самую функцию отсылки UserLib, это сообщение выведется на печать в таком виде:

"The Server has sent data to the Client"
("Сервер отослал данные Клиенту")

Это происходит потому, что переменные *Me* и *NotMe* имеют противоположные значения.

В функциях UserLib в качестве условия перед операторами *printf* используется переменная *LogFlag*.

```
_cdecl main(int argc, char *argv[])
  {
   int rv;
   WORD        VersionRequested;
   WSADATA     ImplementationDetails;
   SOCKET       ClientSocket;
   SOCKADDR_IN ServerSocketAddress;

   if ((argc <=2) || (argc >4))
     { printf(" usage: %s inet-address port [Log-flag]\n",argv[0]);
      printf("  Log-flag of 1 [0 is default] means show log details\n");
      return(0);
     }

   if (4==argc) LogFlag = atoi(argv[3]);
   else LogFlag = 0;
```

Инициализация программы

Первой частью процедуры main является объявление местных переменных и считывание аргументов командной строки. Если программа вызвана неправильным количеством аргументов, то на печать выводится сообщение об использовании (usage message) и программа прекращает работу.

Если количество аргументов правильное, тогда они считываются в соответствующие переменные. Если опциональные аргументы не указаны, то используются значения аргументов по умолчанию.

Когда вы используете библиотеку WINSOCK, которая используется в настоящем примере, требуются только две местных (локальных) переменных *VersionRequested* и *ImplementationDetails*. Если вы используете MS-DOS, тогда эти две переменные вам не нужны. Значения аргументов будут помещены в эти переменные во время фазы инициализации сети.

Для поддержки оконечного пункта, который будет использоваться клиентом для связи с сетью, используется переменная *ClientSocket*. *ServerSocketAddress* содержит адрес *ServerSocket* (*гнездо сервера*), к которому подключается клиент. Этот адрес поставляется клиентом до вызова функции *connect* (*соединение*).

```
printf("Client Echo Program 1.0\n\n");

    /* Initialize Windows Sockets API and Locate proper DLL      */
   /* by calling WSAStartup Windows API                          */
   /* verify that a proper DLL was found                         */

VersionRequested = MAKEWORD(1,1);
rv = WSAStartup (VersionRequested, &ImplementationDetails);
if (0 != rv ) ErrorStop();
```

Инициализация сети

Весь этот раздел можно опустить, если вы переносите эту программу на MS-DOS и не используете библиотеку WINSOCK. Библиотека PWSOCK.LIB операционной системы PATHWORKS не требует инициализации сети.

Вследствие того, что эта программа выполняется в Windows NT и компонована с библиотекой WINSOCK, сеть должна быть инициализирована вызовом *WSAStartup*. Это запрашивает версию 1.1 Sockets DLL и требует, чтобы детали реализации сети были переданы обратно в структуру *ImplementationDetails*.

Помните, что файл UserLib заново определяет *WSAStartup* в *User_WSAStartup* и поэтому в этой программе вы можете задействовать регистрацию (logging).

Функция *ErrorStop()* вызывается, если *WSAStartup* возвращает ненулевое значение, указывающее на возникновение ошибки. Функция *ErrorStop()* выведет на печать указатель на сообщение об ошибке и после этого остановит программу.

```
/*  Create the Client Socket and verify that it was created */

   ClientSocket = socket (PF_INET,SOCK_STREAM,0);
  if (INVALID_SOCKET == ClientSocket) ErrorStop();

/* connect the Socket to an address and port on the network */
/* address is p1 port is p2 */

ServerSocketAddress.sin_family          = AF_INET;
ServerSocketAddress.sin_addr.S_un.S_addr  = inet_addr(argv[1]);
```

```
ServerSocketAddress.sin_port          = atoi(argv[2]);

rv = connect (ClientSocket,(LPSOCKADDR)&ServerSocketAddress,sizeof(ServerSocketAddress));
if (0 != rv) ErrorStop();
```

Создание оконечного пункта и его регистрация с сетью

Для создания оконечного пункта (endpoint) используется функция *socket()*. Три параметра гнездовой функции указывают, что: (1) создаваемая функция использует протокол Internet; (2) гнездо является поточным гнездом; (3) гнездо не имеет никаких специальных атрибутов. Если функция возвращает -1, что соответствует значению INVALID_SOCKET (недействительное гнездо), вы можете вызвать *ErrorStop()* и прекратить выполнение программы.

Допустим, что гнездо было успешно создано и теперь вы можете подключать гнездо к существующему в сети серверу вызовом функции *connect()*, или же вы можете привязать гнездо к адресу, если вы собираетесь выполнять несоединенные (unconnected) сетевые функции. В этой программе целью является подключение к серверу; поэтому вы вызываете функцию *connect()*.

Функции *connect()* требуется знать сетевой адрес сервера. Для определения этого адреса заполните специальную структуру и передайте указатель структуре с помощью функции *connect()*. Структура, на которую указывает *connect()*, должна быть структурой SOCKADDR, которая не зависит от протокола. Для каждого возможного протокола в сети, такого, как Internet Protocol {IP} или DECnet, существуют различные "оттенки" SOCKADDR.

Структура, которая должна быть заполнена, это SOCKADDR_IN, т.е. вариант Internet структуры SOCKADDR. Должны быть поставлены следующие три поля:

- sin_family — употребляется для индикации типа SOCKADDR [Internet],
- sin_addr — употребляется для индикации адреса Internet,
- sin_port — употребляется для указания порта, к которому необходимо подключиться, с помощью адреса.

Элемент *sin_addr* в структуре SOCKADDR_IN предназначен для реализации возможности принятия ряда типов межсетевых (Internet) адресов. Этот элемент в действительности является UNION (*S_un*) — объединением множества других элементов, которые могут быть использованы для присвоения правильного межсетевого (Internet) адреса. Эта программа предполагает, что правильная последовательность знаков,

представляющих Internet Address (адрес Internet), была передана в параметре *argv[1]*. Эта последовательность знаков передается на функцию *inet_addr*, которая возвращает 31-битовое представление адреса. Для того чтобы разместить его в *sin_addr*, используйте элемент объединения *S_addr*. Полное квалифицированное имя имеет вид *sin_addr.S_un.S_addr*.

SOCKADDR_IN

На рис. 4-1 показана структура SOCKADDR и SOCKADDR_IN.

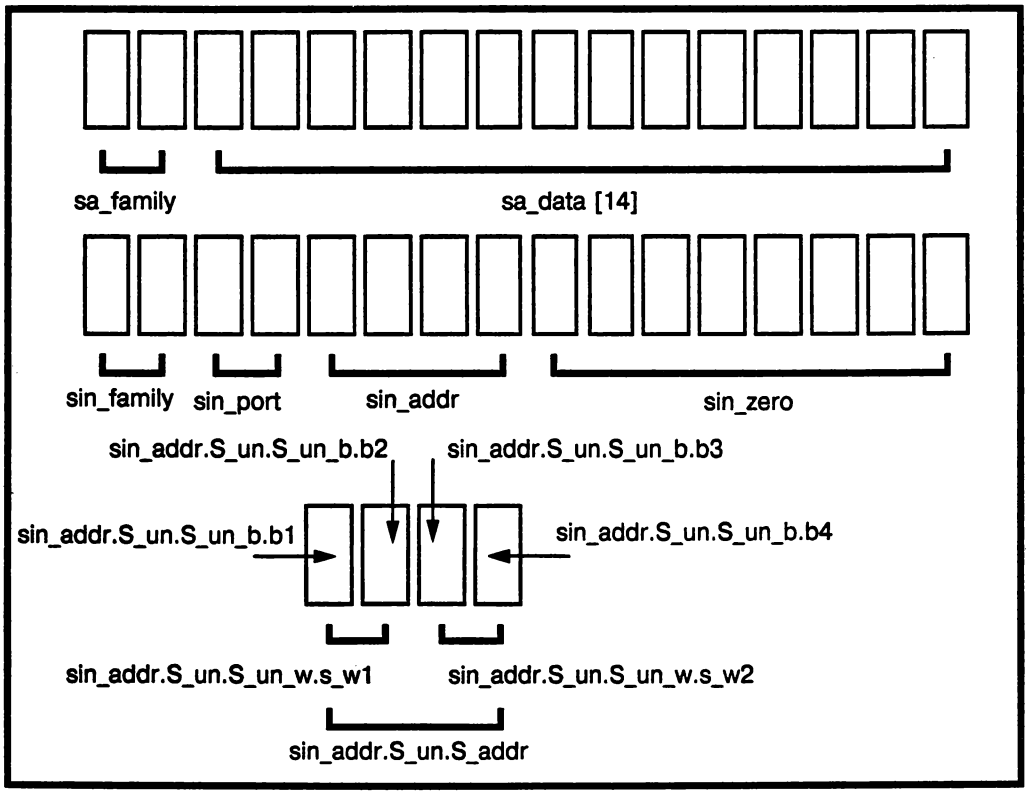


Рис. 4-1. Структура SOCKADDR и SOCKADDR_IN.

```
typedef struct sockaddr {
    u_short      sa_family;
    char         sa_data[14];
} SOCKADDR;

typedef struct sockaddr_in {
    short        sin_family;
    u_short      sin_port;
```

```
    struct in_addr   sin_addr;
    char             sin_zero[8];
} SOCKADDR_IN;

struct in_addr {
   union    {
    struct              { u_char  s_b1, s_b2, s_b3, s_b4; }
                S_un_b;
    struct              { u_short  s_w1, s_w2; }
                S_un_w;
    u_long        S_addr;
    } S_un;
};
  /* Start the Data Processing Loop, wait for quit or stop to exit the loop */

  printf("Enter data to transmit to the server\n");
  UserGetData(ClientSocket);

return(0);
}
```

Обработка данных клиента

После того как клиент подключился к сети и присоединился к серверу, вы начинаете обрабатывать данные клиента. Стандартная подпрограмма осуществляет подсказку пользователю относительно данных, затем посылает данные на сервер. Эта стандартная подпрограмма имеет единственный аргумент — гнездо, которое будет использовано для связи.

```
void UserGetData(SOCKET ClientSocket)
  {
   char buffer[1000];
   int rv;

   while(1==1)
     {
      printf("Client>");
      gets(buffer);

      rv = send (ClientSocket,buffer,strlen(buffer)+1,0);
      if (SOCKET_ERROR == rv) ErrorStop();
```

Считывание и отсылка данных

Когда вызывается подпрограмма, в стеке организуется буфер на 1000 знаков; специальная подсказка *Client>* выводится на печать, и затем пользователю предлагается ввести данные. Для считывания данных с устройства стандартного ввода (stdin), или клавиатуры, используется стандартная функция *gets(char*)*.

Данные отсылаются в сеть с помощью функции *send()*, которая в качестве параметров принимает следующее:

- идентификатор гнезда, переданный в подпрограмму,
- адрес данных, подлежащих отсылке,
- объем данных, подлежащих отсылке плюс один знак, так, чтобы терминатор NUL также был отослан,
- специальные флаги отсылки; обратите внимание на то, что в приводимом примере ни одного такого флага не используется.

Весь этот раздел помещается в бесконечный цикл, так что данные будут поступать для обработки до тех пор, пока не возникнет состояние ошибки или пока команда *exit* не прекратит выполнение подпрограммы.

```
  /* check for the local commands */
if (0==strcmp("exit",buffer))
    { printf("Client sending exit command,Client exiting...\n");
      closesocket(ClientSocket);
      return;
    }
if (0==strcmp("quit",buffer))
    {
      printf("Client sending quit command, Client exiting...\n");
      closesocket(ClientSocket);
      return;
    }
if (0==strcmp("bye",buffer))
   { printf("Client closing socket\n");
    rv =closesocket(ClientSocket);
    if (SOCKET_ERROR == rv) ErrorStop();
    printf("press Return to Exit Client\n");
    getchar();
    return;
   }
if (0==strcmp("clog",buffer))
    {
      printf("Client noticed clog command. LogFlag toggled\n");
      LogFlag = (1==LogFlag) ? 0: 1;
    }
```

Обработка местных команд

После того как данные отправлены на сервер, осуществляется проверка наличия требования выполнения местной команды. На стороне клиента работают следующие местные команды:

- exit и quit

Эти команды заставляют местное гнездо закрыться и затем возвращаются из подпрограммы. После этого функция *main()* отключается. Единственная разница между этими двумя командами состоит в том, что сервер также отключится, если он увидит команду "*Quit*". Так как данные были переданы на сервер до этого шага, сервер увидит данные и также будет иметь возможность отключиться.

- bye

Команда *bye* заставляет клиента закрыть гнездо, однако не выводит клиента из работы. Это моделирует ошибку программирования, при которой клиент неожиданно закрывает свой оконечный пункт (*Endpoint*). Сервер должен быть в состоянии точно обработать эту ошибку. Когда пользователь нажимает <Return> на стороне клиента, клиент отключается от работы.

- clog

Команда *clog* позволяет вам изменить состояние переменной регистрации клиента, когда выполняется программа клиента. Данные *clog* также посылаются на сервер, однако сервер обращается с ними как с нормальными данными, и они не вызывают на сервере специальной обработки.

- Команды сервера вводятся на стороне клиента и передаются на сервер.

Существует множество команд сервера, которые могут быть введены на стороне клиента. Эти команды направляются к серверу и вызывают соответствующую функцию на стороне сервера. Никакой специальной обработки данных клиента не происходит.

```
rv = recv (ClientSocket,buffer,80,0);
if (SOCKET_ERROR == rv) ErrorStop();

printf("socket:%03d>[%03d]",ClientSocket,rv);
output(rv,buffer);  // output is used because buffer may contain NUL characters

  }
}
```

Прием данных сервера

После того как данные посланы на сервер, сервер обрабатывает данные и посылает их обратно по тому же самому соединению. Клиент ожидает этих данных от сервера. Для считывания этих данных с сервера клиент использует функцию *recv()*. Функция *recv()* принимает следующие аргументы:

- Гнездовой идентификатор.
  Это тот же самый идентификатор, который был использован в функции отсылки, и тот же самый, который был передан в подпрограмму;
- Адрес буфера;
- Объем буфера.
  Этот объем произвольно установлен на 80 знаков;
- Специальные флаги. В этом примере не используются.

Значение возврата от функции *recv()* — это число знаков, только что считанное из сети. Эти данные посылаются на специальную функцию под названием *output* (*вывод*), которая используется для вывода данных на экран. *Output* очень похожа на *puts*, за исключением того, что она будет приводить все знаки, которые не могут быть выведены на печать. Так, последовательность знаков "hello\n", которая имеет пять знаков, которые могут быть выведены на печать, и два знака, которые не могут быть выведены на печать — это \n и терминатор NUL — при выводе на печать будет иметь следующий вид:

output("hello\n",7)

hello\010\000

Знаки, которые не могут быть выведены на печать, появляются в виде своих десятичных эквивалентов.

После того как данные, принятые при помощи команды *recv*, приняты, снова осуществляется переход к подсказке "Client>" и начинается новый цикл данных.

СЕРВЕР В РЕЖИМЕ "ЭХО" (С ПОЯСНЕНИЯМИ)

```
#include <winsock.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "Userlib.h"

char * Me = "Server";
char * NotMe   = "Client";
int    WaitFlag;
int    LogFlag;
```

Заголовочные файлы

Для клиента и для сервера используются одни и те же заголовочные файлы, с тем лишь исключением, что сервер имеет дополнительный стандартный заголовочный файл в С, потому что он использует функции, определенные в *ctype.h*. В нашем примере используется заголовочный файл WINSOCK.H, потому что файл создается как консольная программа Windows NT. Если эта программа переносится на MS-DOS, то тогда, возможно, потребуется выбирать другой заголовочный файл.

Глобальные переменные

Отметим, что две переменные — *Me* и *NotMe* — имеют обратные значения при поступлении от клиента. Это позволяет функциям в файле включения *UserLib* работать нормально.

Сервер имеет две флаговых переменных:

- WaitFlag — используется для принятия решения о необходимости ожидания до обработки данных клиента.
- LogFlag — используется для принятия решения, следует ли выводить на печать сообщения регистрации.

```
_cdecl main(int argc, char *argv[])
 {
 int rv;
 int ServerStopFlag;
 unsigned short port;

 WORD        VersionRequested;
 WSADATA      ImplementationDetails;
 SOCKET       ServerSocket;
 SOCKET       ClientSocket;
 SOCKADDR_IN ServerSocketAddress;
 SOCKADDR_IN ClientSocketAddress;

 if ((argc<=1)||(argc>4))
   {printf("Usage: %s port-number [wait-flag] [log-flag]\n",argv[0]);
    printf(" if port = 0 then BIND will choose port number\n");
    printf(" if wait-flag is 1 [default is 0], then user must hit return to recv\n");
    printf(" if log-flag is 1 [default is 0], then detail log messages appear\n");
    return(0);
   }

 port     = atoi(argv[1]);
 if (2<argc)      WaitFlag = atoi(argv[2]);
 else            WaitFlag = 0;
 if (3<argc)      LogFlag  = atoi(argv[3]);
 else            LogFlag  = 0;
```

Инициализация программы

Сервер, как и клиент, должен инициализировать сеть при работе под библиотекой Windows. Первые две объявленные переменные используются командой WSAStartup для осуществления инициализации сети.

Отметим, что у сервера имеются две гнездовые переменные. Одна должна применяться для значения оконечного пункта, который используется для прослушивания новых клиентов. Другая используется для значения оконечного пункта текущего клиента.

Кроме того, отметим, что существуют две переменные SOCKADDR_IN. Одна используется для установления адреса оконечного пункта сервера. Другая заполняется адресом текущего клиента, когда он принимается сервером.

Действия сервера на этапе инициализации программы имеют во многом ту же логику, что и клиента. Если присутствует неправильное количество аргументов, сервер выдает сообщение об использовании (a usage message) и выходит из режима обработки.

Если присутствует правильное количество аргументов, то аргументы считываются в соответствующие переменные. Если опциональные аргументы отсутствуют, то используются значения по умолчанию.

```
printf("Server Echo Program 1.0\n\n");
VersionRequested = MAKEWORD(1,1);
rv = WSAStartup (VersionRequested, &ImplementationDetails);
if (0 != rv ) ErrorStop();
```

Инициализация сети

Это действительный код программы для инициализации сети в библиотеке WINSOCK. Если эта программа переносится на другую платформу, которая не имеет WINSOCK, этот раздел не нужен.

```
ServerSocket = socket (PF_INET,SOCK_STREAM,0);
if (INVALID_SOCKET == ServerSocket) ErrorStop();

ServerSocketAddress.sin_family              = AF_INET;        // InterNet
ServerSocketAddress.sin_addr.S_un.S_addr    = INADDR_ANY;     // Any Address
ServerSocketAddress.sin_port                = port;

rv = bind (ServerSocket,(LPSOCKADDR)&ServerSocketAddress,sizeof(ServerSocketAddress));
if (0 != rv ) ErrorStop();
rv = listen (ServerSocket,5);
if (0!= rv) ErrorStop();
```

Подготовка гнезда сервера

Этот раздел программы выполняет следующие функции:

- создает оконечный пункт гнезда сервера;
- соединяет оконечный пункт с конкретным портом;
- адрес INADDR_ANY позволяет серверу осуществить привязку к любому адресу в системе "клиент-сервер" с множественной адресацией.

Примечание: Система с множественной адресацией (multihomed) имеет множественные сетевые карты (cards) и обычно действует как маршрутизатор межсетевых протоколов (IP). Один адрес TCP/IP привязывается к одной плате. Второй, с отличным сетевым номером, привязывается к другой плате.

Он будет привязываться только к одному из адресов. Если вы не находитесь в системе с множественной адресацией, то он будет осуществлять привязку к единственному доступному адресу — адресу сервера;

- функция *bind()* осуществляет привязку адреса, определенного в структуре *ServerAddress*, к конкретному гнезду;
- функция *listen()* устанавливает очередь клиентов. Когда клиент пытается соединиться с сервером, он получает автоматическое подтверждение и не блокируется в функции *connect()*. Если сервер не осуществляет прослушивание, то клиент получает от *connect* состояние сетевой ошибки. Хотя функция *listen* позволяет клиенту продолжать обработку, она не устанавливает соединение. Если клиент начинает посылать данные на сервер до того, как осуществлено соединение, данные будут буферироваться на стороне клиента до тех пор, пока выходной буфер клиента не будет заполнен. Так как еще нет соединения с сервером, то как только выходной буфер клиента заполнится, клиентская программа блокируется до тех пор, пока сервер не примет клиента. Затем данные могут быть переданы из выходного буфера клиента на входной буфер сервера.

```
ServerStopFlag = 0;
while (0==ServerStopFlag)
    {
     ClientSocket = accept (ServerSocket,(LPSOCKADDR) &ClientSocketAddress,NULL);
     if (INVALID_SOCKET == ClientSocket) ErrorStop();
     ServerStopFlag = UserProcessClient(ClientSocket) ;
    }
printf("Exiting the Server ....\n");
return(1);
}
```

Обработка клиентов

Этот раздел программы принимает клиента и обрабатывает передаваемую и/или принимаемую информацию. Если имеется клиент, ожидающий в этот момент прослушивания в очереди, то функция *accept()* не блокируется. Когда *accept* осуществляет возвращение, связь по установленному соединению осуществляется немедленно. Если текущих клиентов в очереди на прослушивание нет, то *accept* блокируется и не осуществляет возвращения, пока не появится клиент на прослушивание.

Если *accept* осуществляется успешно, то она возвращает новое гнездо, которое имеет аналогичные атрибуты, что и гнездо, которое прослушивало клиента. Для связи с клиентом, который только что подключился, используется новое гнездо, а старое гнездо становится доступным для принятия другого клиента. В множественной системе "клиент-сервер" сервер имеет возможность одновременно принимать нескольких клиентов. В этом случае гнезда различных клиентов хранятся в массиве или другой структуре, для того чтобы дать возможность серверу обрабатывать одновременно нескольких клиентов.

В этом примере сервер в любой момент времени обрабатывает только одного клиента и не будет принимать другого клиента, пока первый клиент не отключится. Клиент обрабатывается подпрограммой *UserProcessClient*. Подпрограмма возвращает число, которое указывает, следует ли серверу продолжать обрабатывать информацию клиентов. Если *UserProcessClient* возвращает 0, то начальное условие *while* будет оставаться истинным и будет осуществлен прием и обработка другого клиента. Если подпрограмма возвращает ненулевую величину, то условие *while* оказывается ложным и функция *main* отключится.

```
int UserProcessClient(SOCKET ClientSocket)
  {
   char      buffer[100];
   int       rv;
   while (1==1)
      {
      if (1==WaitFlag)
        {printf("  Press Return to receive data from %s\n",NotMe);
         getchar();
        }
      rv = recv (ClientSocket,buffer,80,0);
```

Прием данных от клиента

Эта программа предназначена для приема данных от клиента, их обработки и последующей отсылки обратно обработанных данных.

Если установлен флаг *WaitFlag*, то пользователю предлагается нажать <Return> на сервере до приема данных от клиента. После того как пользователь нажмет <Return>, функция *recv* вызывается для считывания до 80 знаков из входного буфера. Эти данные поступают в буфер массива знаков (character array buffer). Массив знаков имеет максимальный размер 100 знаков, поэтому вы можете принять максимум 100 знаков. Однако программа закодирована жестким кодом на 80 знаков.

```
if (SOCKET_ERROR == rv)
  switch(WSAGetLastError())
    { case(WSAECONNRESET):
      printf("Connection Reset, Listening for a New Client..\n");
      closesocket(ClientSocket);
      return(0);  // signals to accept a new client
     default: ErrorStop();
    }
if (0==rv)
    {printf("Zero Length Data interpreted as a network disconnect\n");
     printf("Connection Reset, Listening for a New Client..\n");
     closesocket(ClientSocket);
     return(0);   // signals to accept a new client
    }

printf("socket:%03d>[%03d]",ClientSocket,rv);
output(rv,buffer);  // output is used because buffer may contain NUL characters
```

Проверка ошибок

Следующим шагом, который выполняется во время обработки данных клиента, является проверка ошибок при приеме. Функция *recv()* возвращает количество принятых знаков. Если имела место ошибка, возвращается значение SOCKET_ERROR (-1). Для точного определения ошибки может быть вызвана функция *GetLastError*. Код программы осуществляет проверку на предмет установления того, была ли возвращена ошибка WSACONNRESET. Это указывает на то, что клиент закрыл свою сторону гнезда. Правильным ответом сервера является закрытие его стороны гнезда и прием другого клиента. Возвращенное значение 0 указывает, что сервер должен принимать другого клиента.

Если сервер принимает последовательность нулевой длины, то это интерпретируется как ошибка. Это может иметь место, если клиент выходит из обработки, не закрывая свое гнездо на DOS-компьютере. Сервер не обязательно принимает запрос на разъединение, так как DOS по-прежнему считает, что гнездо открыто. Если возникает эта ситуация, процедура закрывает серверную сторону гнезда и принимает для обработки нового клиента. Значение возврата 0 указывает, что сервер должен принять другого клиента.

Если не возникает ни одной из этих ошибок, то данные, принятые сервером, выводятся на печать. Для вывода данных на дисплей используется функция *output. Output* используется потому, что она печатает "данные". Если знак имеет значение ASCII 31 или меньше, то знак печатается как *nnn*, где *nnn* — десятичный эквивалент знака.

```
if (0==strcmp("exit",buffer))
  { printf("Client sent Exit command, Waiting for another client\n");
    closesocket(ClientSocket);
    return(0);    // signals to accept a new client
  }
if (0==strcmp("quit",buffer))
  {
    printf("Client sent quit command, Server exiting...\n");
    closesocket(ClientSocket);
    return(1);   // signals to exit the server
  }
if (0==strcmp("slog",buffer))
  {
    printf("Client sent slog command. LogFlag toggled\n");
    LogFlag = (1==LogFlag) ? 0: 1;
  }
if (0==strcmp("swait",buffer))
  {
    printf("Client sent swait command. WaitFlag toggled\n");
    WaitFlag = (1==WaitFlag) ? 0: 1;
  }
```

Проверка команд сервера

Этот раздел сервера проверяет данные, которые только что были переданы от клиента, на предмет установления, есть ли там какие-либо вложенные команды, предназначенные для сервера. Команды SLOG и SWAIT используются для переброса текущего состояния флагов *LogFlag* и *WaitFlag* на сервере. Для оповещения сервера о том, что клиент прекращает работу, клиент посылает на сервер команду EXIT. Затем сервер закрывает свою сторону соединения и готовится к приему нового клиента.

Команда QUIT используется для указания серверу, что он должен выйти из цикла сервера и не должен больше обрабатывать никаких клиентов. Когда эта подпрограмма возвращает 1, сервер останавливается.

```
UserWork(buffer);
rv = send (ClientSocket,buffer,strlen(buffer)+1,0);
if (SOCKET_ERROR == rv)
  switch(WSAGetLastError())
    { case(WSAECONNRESET):
      printf("Connection Reset, Listening for a New Client..\n");
```

```
        return(0); // signals to accept a new client
        default: ErrorStop();
      }
    }
}
```

Обработка данных клиента

Функция *UserWork(char*)* принимает последовательность знаков и добавляет 1 к каждой букве последовательности. Это используется для имитирования определенного вида работы, которую сервер выполняет по поручению клиента. Когда эта работа выполнена, то для передачи данных обратно к клиенту используется функция *send()*. Если при попытке передать данные возникает состояние ошибки, тогда процедура проверяет ситуацию, для того чтобы удостовериться, что ошибка была WSAECONNRESET. Если это так, то она *принимает* нового клиента. Если же имеет место другой тип ошибки, то процедура выводит на печать сообщение, указывающее то, что произошло, и затем останавливает сервер.

USERLIB.C (С ПОЯСНЕНИЯМИ)

```
#include <winsock.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "Userlib.h"

#undef WSAStartup
#undef socket
#undef bind
#undef listen
#undef accept
#undef connect
#undef recv
#undef send
```

Эти операторы *#undef* используются для того, чтобы убедиться, что гнездовые функции не были переадресованы в библиотеку. Если операторы *#undef* не используются, то тогда, когда вы выполняете команду *send()* в рамках функции *User_send()*, вызов будет переадресован обратно и будет снова осуществляться вызов функции *User_send()* внутри самой себя. Это ведет к бесконечной рекурсии и, естественно, является грубой ошибкой.

```
void UserErrorMessage(char *message)
      {
       char buffer[100];
       char *Msg;
       int err;
       if (1==LogFlag) return;
       err = WSAGetLastError();
       printf("%s",message);
       switch(err)
         {
          case(WSAEINTR              ): Msg="WSAEINTR        ";
                    break;
          case(WSAEBADF              ): Msg="WSAEBADF        ";
                    break;
          case(WSAETIMEDOUT          ): Msg="WSAETIMEDOUT    ";
                    break;
          case(WSAECONNREFUSED       ): Msg="WSAECONNREFUSED ";
                    break;
          .
          .
          .

          case(WSANO_DATA            ): Msg="WSANO_DATA      ";
                    break;
          default:sprintf(buffer,"Unknown: Error (%d)",err);
              Msg=buffer;
              break;

         }
  printf("      Error:%s",Msg);
}
```

Эта функция используется для вывода на печать последовательности знаков в ответ на специфичную ошибку WSA. Она не прекращает выполнение программы.

```
void ErrorStop(
  {
   char buffer[100];
   char *Msg;
   int err;
   err = WSAGetLastError();
   switch(err)
   {
          case(WSAEINTR              ): Msg="WSAEINTR        ";
                    break;
          case(WSAEBADF              ): Msg="WSAEBADF        ";
                    break;
          case(WSAETIMEDOUT          ): Msg="WSAETIMEDOUT    ";
                    break;
```

```
            case(WSAECONNREFUSED          ): Msg="WSAECONNREFUSED        ";
                    break;
            case(WSANO_DATA             ): Msg="WSANO_DATA             ";
                    break;
  .
  .
  .
   default: sprintf(buffer,"Unknown: Error(%d)",err);
       Msg=buffer;
      break;
    }
   printf("Fatal: (%s)\n",Msg);
   exit(err);
}
```

Эта функция аналогична предыдущей функции, однако она прекращает выполнение программы после вывода сообщения на печать.

```
void output(int count, char *buffer)
  {
  while (count > 0)
    {
    if (31>*buffer)   printf("\\%03d",*buffer);
    else              putchar(*buffer);

    buffer++;
    count—;
    }
  printf("\n");
  }
```

Для вывода на печать последовательности знаков используется функция *output*. Вы передаете этой функции подсчет знаков и указатель на последовательность. Для каждого знака последовательности, который имеет значение ASCII менее 31, знак выводится на печать в виде *nnn*, где *nnn* — десятичный эквивалент знака.

```
void UserShowAddress(LPSOCKADDR_IN AddressPointer)
   {
      if (1==LogFlag) return;
      printf("       [AF_INET family is %d]\n",AF_INET);
      printf("            sin_family: %d\n",AddressPointer->sin_family);
      printf("            sin_port:   %d\n",AddressPointer->sin_port);
      printf("            sin_addr: %d.%d.%d.%d\n",
        AddressPointer->sin_addr.S_un.S_un_b.s_b1,
        AddressPointer->sin_addr.S_un.S_un_b.s_b2,
        AddressPointer->sin_addr.S_un.S_un_b.s_b3,
        AddressPointer->sin_addr.S_un.S_un_b.s_b4);
   }
```

Функция *UserShowAddress* принимает единственный параметр — указатель на межсетевой SOCKADDR. На печать выводятся различные члены этой структуры.

```
void UserWork(char *temp)
  {
   for(;'\0'!=*temp;temp++) (*temp)++;
  }
```

"Переназначенные" функции гнезд

Каждая из следующих функций представляет собой переназначенную (redirected) гнездовую функцию. Гнездовые функции, вызванные в ECHO.C и CLIENT.C, были переназначены в эти функции с помощью макрооператоров #define. Каждая из этих функций просто проверяет глобальные переменные *LogFlag*, *Me* и *NotMe* для вывода на печать различных сообщений. Затем они вызывают первоначальную гнездовую функцию и передают значения обратно на вызывающую программу (caller).

```
int UserLog_WSAStartup(WORD VersionRequested, WSADATA *ImplementationDetailsPointer)
  {
   int err;

   if (1==LogFlag) printf("Calling WSAStartup() to locate a Windows DLL\n");

   err = WSAStartup(VersionRequested, ImplementationDetailsPointer);

   if (0!=err)
    {
      if (1==LogFlag) {
         printf("      Couldn't locate a Windows Sockets DLL\n");
         printf("      WSAStartup Failed (Error Code = %d)\n",err);
                }
      return(err);
    }
   if (1==LogFlag)
   {
     printf   WSAStartup has returned ImplementationDetails:\n");
     printf("  wVersion:     (%ld)\n",ImplementationDetailsPointer->wVersion);
     printf(" wHighVersion: (%ld)\n",ImplementationDetailsPointer->wHighVersion);
     printf("   szDescription:'%s'\n",ImplementationDetailsPointer->szDescription);
     printf("    szSytemStatus:'%s'\n",ImplementationDetailsPointer->szSystemStatus);
     printf("   iMaxSockets:  (%d)\n",ImplementationDetailsPointer->iMaxSockets);
     printf(    iMaxUdpDg:    (%d)\n",ImplementationDetailsPointer->iMaxUdpDg);
     printf("   lpVendorInfo  (%p)\n",ImplementationDetailsPointer->lpVendorInfo);
     printf("\n\n");
    }
    return(0);
   }
```

```
SOCKET UserLog_socket(int AddressFormat,int Type,int Protocol)
 {
  SOCKET ServerSocket;
  if (1==LogFlag) printf("Calling socket() to Create a %s Socket\n",Me);

  ServerSocket = socket(AddressFormat,Type,Protocol);

  if (INVALID_SOCKET == ServerSocket) UserErrorMessage("Socket creation Failed\n");
  else if (1==LogFlag) printf(" %s Socket (%d) has been created\n\n",Me,ServerSocket);

  return(ServerSocket);
 }

int UserLog_bind(SOCKET ServerSocket,LPSOCKADDR AddressPointer, int len)
  {
   int err;
   if (1==LogFlag) printf("Calling bind() to Bind the socket\n");

   err = bind(ServerSocket,(LPSOCKADDR)AddressPointer,len);

   if (0 != err) UserErrorMessage("Socket binding failed\n");
   else
     {if (1==LogFlag) printf("       Bind succeeded\n");
      UserShowAddress((LPSOCKADDR_IN)AddressPointer);
      if (1==LogFlag) printf("\n");
     }
   return(err);

   }
int UserLog_listen(SOCKET ServerSocket, int MaxConnections)
  {
    int err;
    if (1==LogFlag) printf("Calling listen(Socket,MaxConnections)\n");

    err = listen(ServerSocket,MaxConnections);

    if (0 != err) UserErrorMessage("Failed to Change to a listen Socket\n");
    else  if (1==LogFlag) printf("Successfully changed to a listen socket\n\n");

    return(err);
  }

SOCKET UserLog_accept(SOCKET ListenSocket,LPSOCKADDR AddressPointer, int *lenPointer)
  {
   if (1==LogFlag) printf("Calling accept(ListenSocket,AddressPointer,length)\n");
   if (1==LogFlag) printf("This function will BLOCK until a client is available\n");

   ListenSocket = accept(ListenSocket,(LPSOCKADDR)AddressPointer,lenPointer);

   if (INVALID_SOCKET == ListenSocket) UserErrorMessage("Socket accept failed\n");
```

```
    else if (1==LogFlag) printf("accept succeeded: A client is available\n\n");

    return(ListenSocket);
   }

int UserLog_connect(SOCKET ServerSocket,LPSOCKADDR AddressPointer, int len)
   {
    int err;
    if (1==LogFlag) printf("Calling connect(ServerSocket,AddressPointer,length)\n");
    UserShowAddress((LPSOCKADDR_IN)AddressPointer);

    err = connect(ServerSocket,(LPSOCKADDR)AddressPointer,len);

    if (0!= err) UserErrorMessage("Socket connect failed\n");
    else if (1==LogFlag) printf("Connect succeeded: A %s is available\n\n",NotMe);

    return(err);
   }

 int UserLog_recv(SOCKET ClientSocket,char *buffer,int sBuffer,int flags)
   {
    int err;
    if (1==LogFlag) printf("Calling recv(Socket,buffer,SizeOfBuffer,Flags)\n");

    err = recv(ClientSocket,buffer,sBuffer,flags);

    if (SOCKET_ERROR == err) UserErrorMessage("Socket recv() failed\n");
    else
      if (0 == err) if (1==LogFlag) printf("Zero Characters Read\n");
      else if (1==LogFlag)
          printf("%d characters read from the %s\nData:%s\n",err,NotMe,buffer);

      return(err);
     }

int UserLog_send(SOCKET ClientSocket,char *buffer,int sBuffer,int flags)
   {
      int err;
      if (1==LogFlag) printf("Calling send(Socket,buffer,SizeOfBuffer,Flags)\n");

      err = send(ClientSocket,buffer,sBuffer,flags);

      if (SOCKET_ERROR == err) UserErrorMessage("Socket send() failed\n");
      else
         if (1==LogFlag)
            printf("%d characters sent to the %s\nData: %s\n",err,NotMe,buffer);

     return(err);
     }
```

СОЗДАНИЕ ГОТОВЫХ К ВЫПОЛНЕНИЮ ПРОГРАММ

Компилирование под NT SDK

Для компилирования и компоновки исходных файлов с помощью набора средств разработчика программного обеспечения Windows NT был использован следующий формирующий файл:

```
!include <ntwin32.mak>
cflags=$(cflags) -DDEVL=1 -D_MT=1 -Gz
all: server.exe client.exe
server.exe: server.obj userlib.obj $(link) $(linkdebug) $(conflags) -out:$*.exe $**  $(conlibsmt)
wsock32.lib
client.exe: client.obj userlib.obj $(link) $(linkdebug) $(conflags) -out:$*.exe $** $(conlibsmt)
wsock32.lib
client.obj: client.c userlib.h  $(cc) $(cflags) $(cvarsmt) $(cdebug) client.c
server.obj: server.c userlib.h  $(cc) $(cflags) $(cvarsmt) $(cdebug) server.c
userlib.obj: userlib.c userlib.h  $(cc) $(cflags) $(cvarsmt) $(cdebug) userlib.c
```

Этот формирующий файл описывает, как создать следующие три файла .OBJ:

- Client.OBJ — зависит от Client.c и Userlib.h,
- Server OBJ — зависит от Server.c и Userlib.h,
- Userlib.OBJ — зависит от Userlib.c и Userlib.h.

Два файла .EXE создаются следующим образом:

- SERVER.EXE — зависит от server.obj и userlib.obj,
- CLIENT.EXE — зависит от client.obj и userlib.obj.

Компонуйте как стандартную консольную программу *$(conflags)* и осуществляйте компоновку со следующими библиотеками:
— консольные библиотеки со множественной обработкой *$(conlibsmt)*,
— библиотека гнезд Windows WSOCK32.LIB.

ИСПОЛЬЗОВАНИЕ VISUAL C++

Создание файла нового проекта для SERVER.EXE

Для создания нового проекта в Visual C++ используйте выбор меню Project/New (см. рис. 4-2). Эта команда позволяет вам создать новый формирующий файл, который генерирует .EXE. Вам нужно определить исходные файлы, которые будут встроены в проект, а Visual C++ будет сканировать файлы для определения того, от каких заголовочных файлов зависит исходный файл.

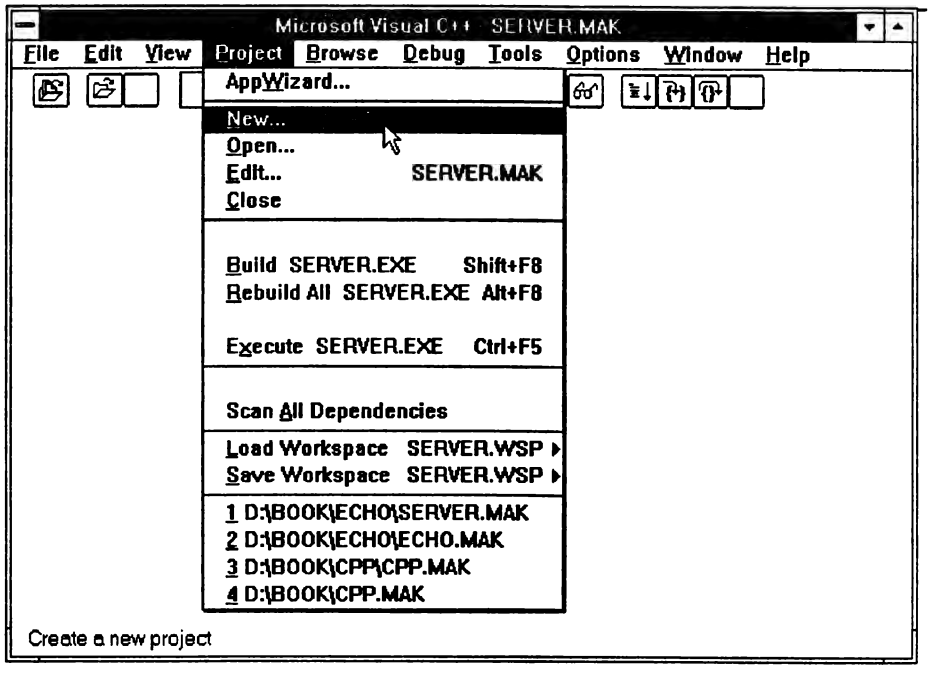


Рис. 4-2. Использование выбора меню Project/New.

Определение деталей проекта

Когда вы выбрали элемент меню Project/New, диалоговое окно Project/New выводится на дисплей (см. рис. 4-3). Используйте этот диалог для идентификации типа прикладной программы, которую вы создаете, и имени формирующего файла. Вы создаете консольную прикладную программу, которая называется SERVER. В этом проекте вы не будете использовать базовые классы фирмы Microsoft (Microsoft Foundation Classes). Клавиша Browse используется для поиска существующего формирующего файла с целью его открывания. Вы создаете новый формирующий файл под названием SERVER.MAK.

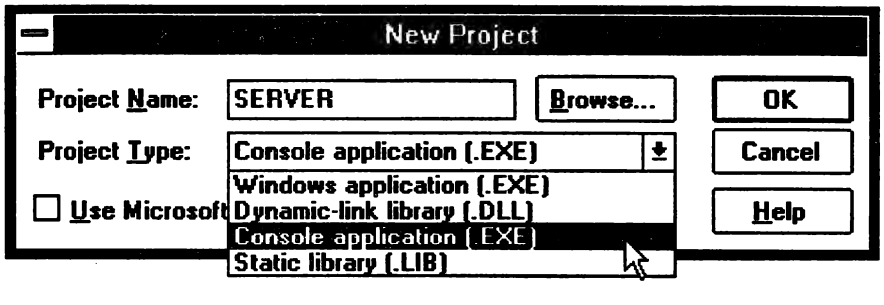


Рис. 4-3. Диалоговое окно нового проекта.

Добавление к проекту исходных файлов

После того как вы полностью заполните диалоговое окно Project New, на дисплее появится диалоговое окно Edit (см. рис. 4-4). Используйте этот диалог для добавления ваших исходных файлов к проекту. Все файлы .c, .cpp и .cxx в текущем каталоге появляются в верхнем окне списка. Выберите файлы, которые вы хотите добавить к вашему проекту, и щелкните мышью по клавише Add. Файлы, которые вы выбираете, появятся в нижнем окне списка. Проект SERVER включает в себя исходные файлы SERVER.C и USERLIB.C. Когда вы включили в проект все файлы, щелкните мышью по клавише Close.

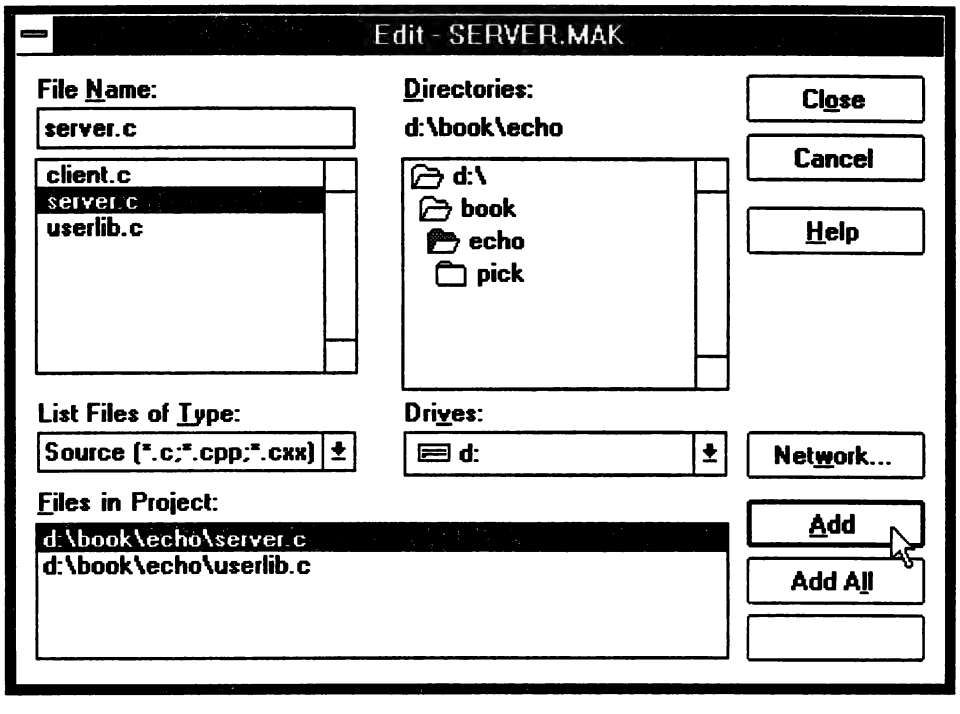


Рис. 4-4. Диалоговое окно редактирования.

Установка опций проекта

Как только вы создали формирующий файл проекта, вам необходимо установить опции компилирования и компоновки для проекта. Используйте элемент меню Options Project для установки опций проекта (см. рис. 4-5).

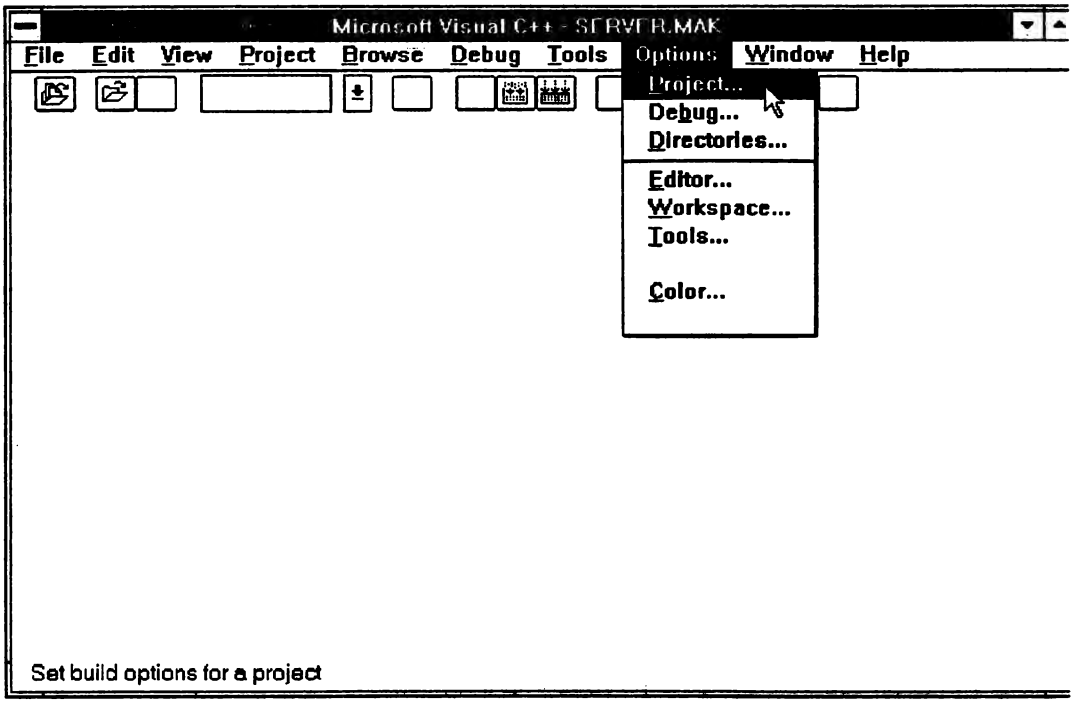


Рис.4-5. Элемент меню Options Project.

Диалоговое окно Project Options

Для установки опций для вашего формирующего файла используйте диалоговое окно Project Options (см. рис. 4-6). Когда вы создавали новый проект, вы осуществили идентификацию консольной прикладной программы (Console Application). Три клавиши в этом диалоговом окне используются для установки опций компилирования, компоновки и ресурсов. Приемлемы опции компилирования по умолчанию. Для консольной прикладной программы опции ресурсов не подходят. Вам необходимо установить опции компоновки (Link) для включения WSOCK32.LIB во время фазы компоновки, так чтобы функции, которые вы вызываете, были разрешены.

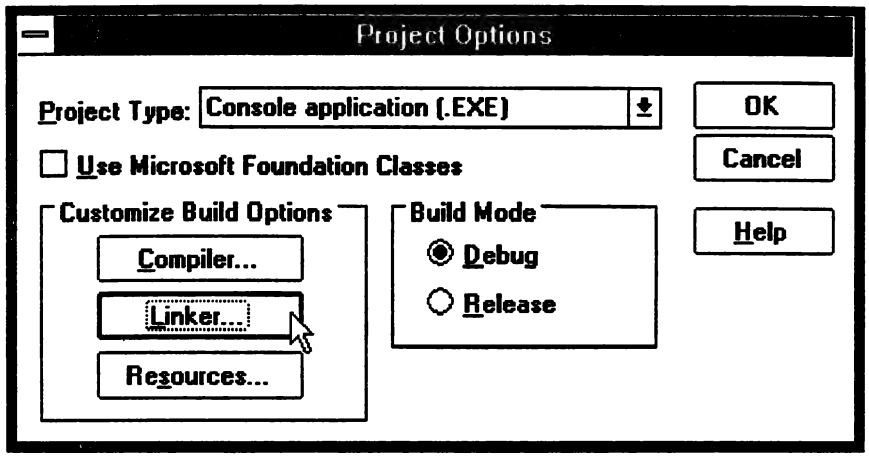


Рис. 4-6. Использование диалогового окна Project Options.

Установка опций компоновщика

Если вы щелкнете мышью по клавише Linker..., на экране появится диалоговое окно Linker Options (см. рис.4-7). Левая часть этого диалогового окна используется для идентификации классов опций компоновщика, которые можно устанавливать. Здесь вы устанавливаете новые опции Input. Опции Input позволяют вам определить дополнительные библиотеки, с которыми можно осуществлять компоновку. Для того чтобы осуществить компоновку с библиотеками гнезд, был введен WSOCK32.LIB.

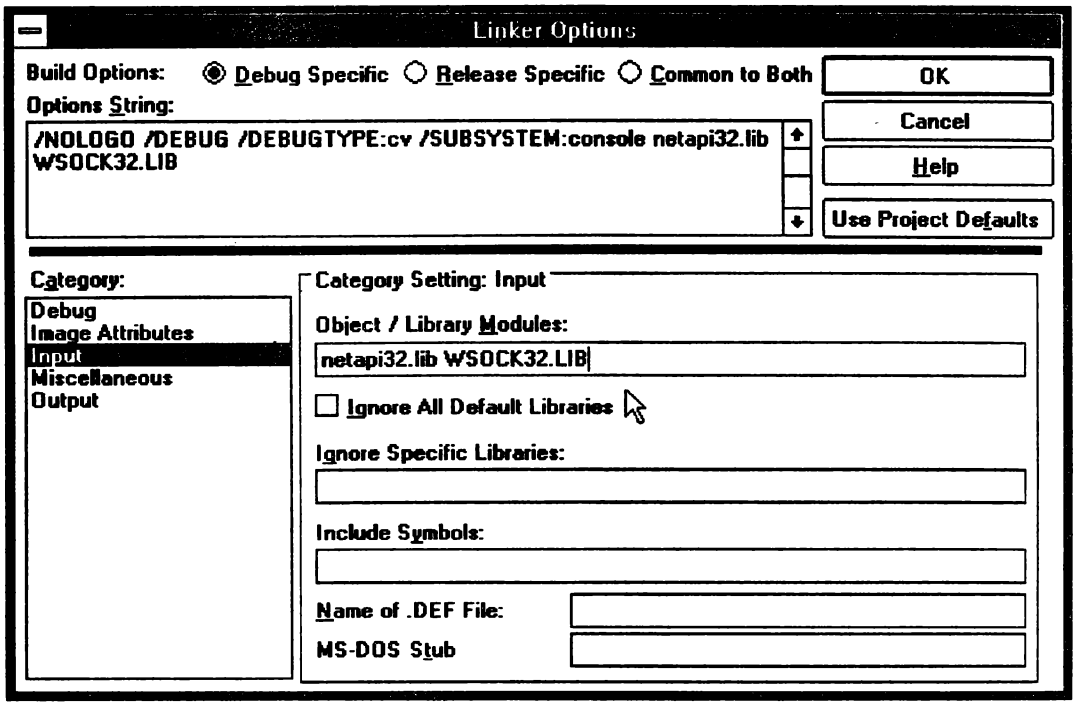


Рис. 4-7. Диалоговое окно Linker Options.

Построение прикладной программы

Для вызова компилятора и компоновщика щелкните мышью по клавише Build (см. рис. 4-8). На нижеприведенном рисунке показаны ошибки компоновщика, с которыми вы можете столкнуться, если не установите правильные опции компоновщика для включения файла WSOCK32.LIB, как это описано выше.

Формирует готовую к выполнению программу — Компилирует только необновленные файлы, затем компонует

Копирует (и только) текущий файл

Формирует готовую к выполнению программу — Компилирует и компонует все файлы проекта

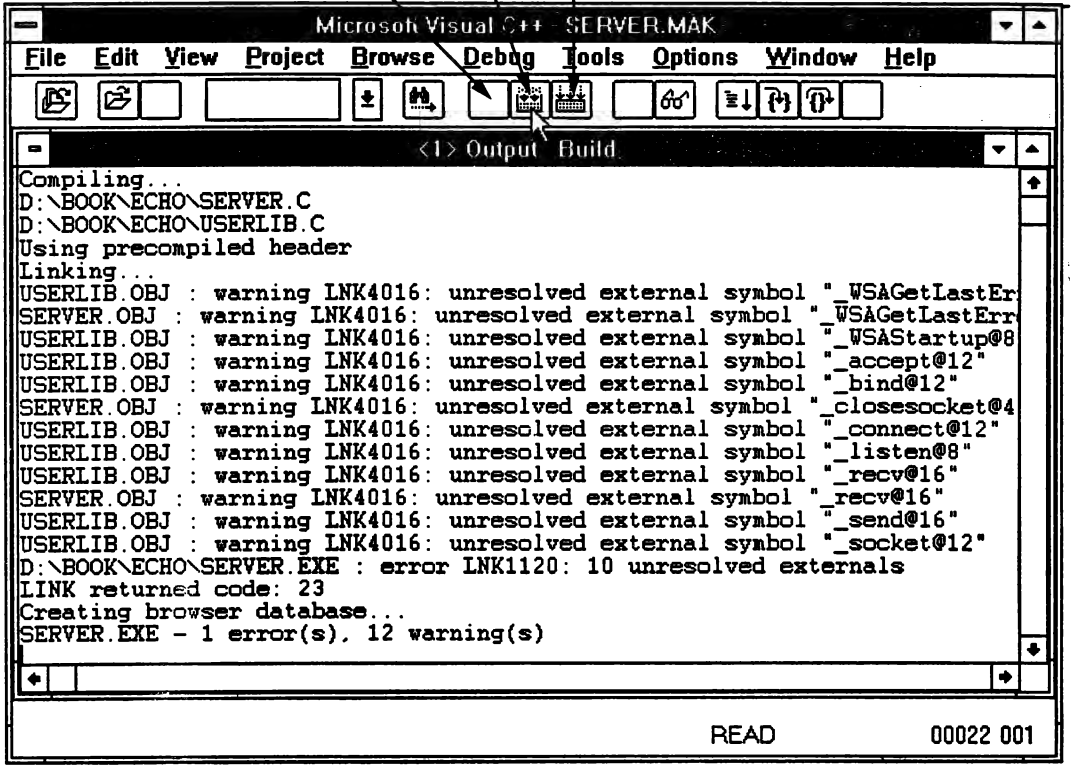


Рис. 4-8. Использование Build для вызова компилятора и компоновщика.

Построение приложения

На рис. 4-9 показаны результаты построения текущего проекта, когда опции компоновщика были установлены для включения библиотеки WSOCK32.LIB.

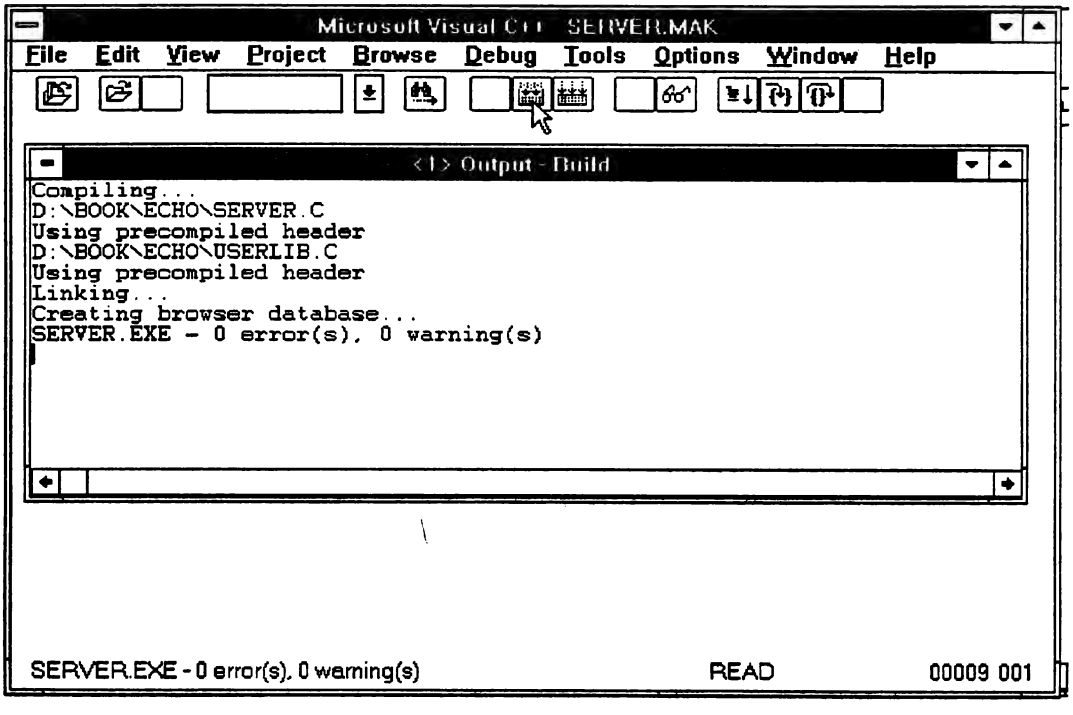


Рис. 4-9. Результаты построения текущего проекта.

Прогон сервера

Для установки аргументов командной строки для сервера используйте элемент меню Options Debug (см. рис. 4-10).

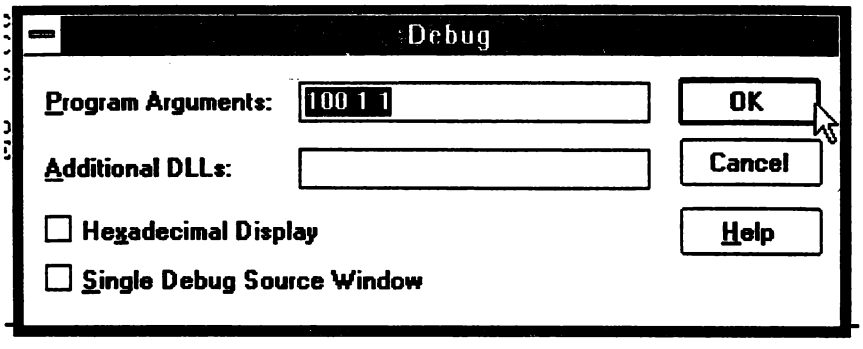


Рис. 4-10. Использование Options Debug для установки аргументов командной строки.

Аргументы командной строки 100 1 1 представляют следующее:

- 100 — порт
- 1 1 — текущие значения *WaitFlag* и *LogFlag*

Старт сервера

Для старта прикладной программы сервера используйте элемент меню Project Execute [SERVER.EXE] (см. рис. 4-11).

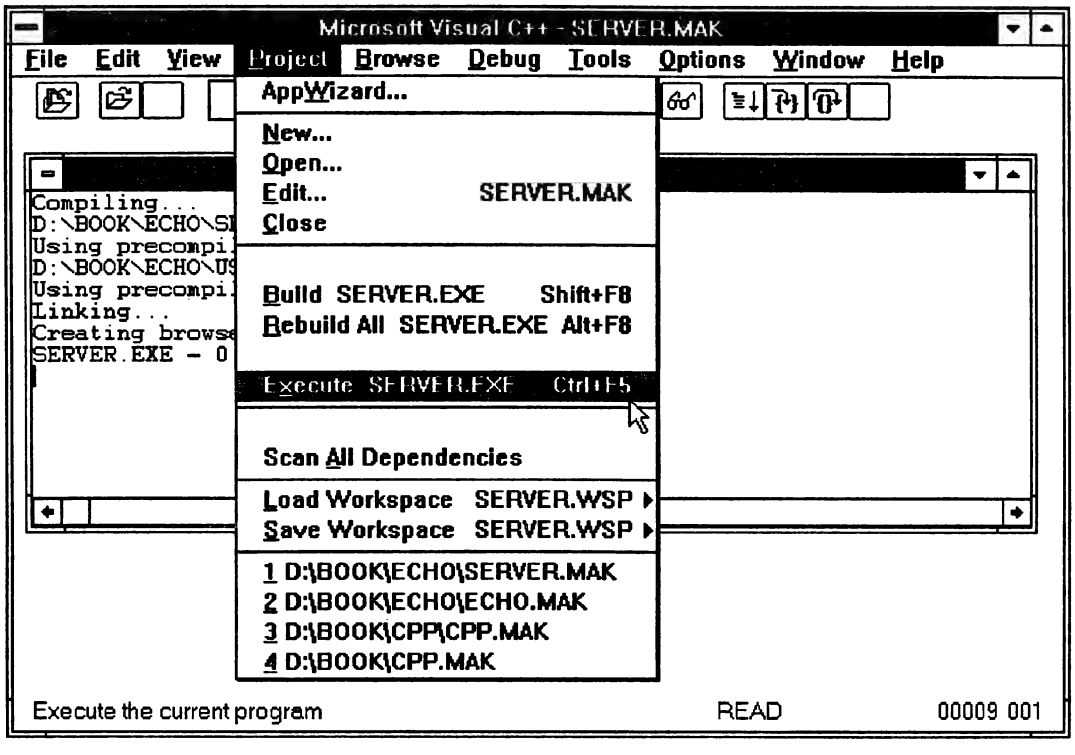


Рис. 4-11. Старт прикладной программы сервера.

Программа сервера

Когда сервер стартует с задействованным флагом log flag, вы увидите дисплей, показанный на рис. 4-12.

```
D:\BOOK\ECHO\SERVER.EXE
Server Echo Program 1.0

        Calling WSAStartup(Version,Details) to locate a Windows DLL
        WSAStartup has returned ImplementationDetails:
          wVersion:     (257)
          wHighVersion: (257)
          szDescription:'Microsoft Windows Sockets Version 1.1.'
          szSytemStatus:'Running.'
          iMaxSockets:  (32767)
          iMaxUdpDg:    (65467)
          lpVendorInfo  (00014C6C)


        Calling socket(AddressFormat,Type,Protocol) to Create a Server Socket
        Server Socket (92) has been created

        Calling bind(Socket,AddressPointer,length) to Bind the socket
        Bind succeeded

        Calling listen(Socket,MaxConnections)
        Successfully changed to a listen socket

        Calling accept(ListenSocket,AddressPointer,length)
        This function will BLOCK until a client is available
```

Рис. 4-12. Старт сервера с задействованным флагом log flag.

Создание файла New Project для CLIENT.EXE

Для создания файла нового проекта для клиента используйте ту же процедуру, которая была использована для создания проекта сервера. Выбирается элемент меню Project New. Проектное имя КЛИЕНТА (CLIENT) определяется в диалоговом окне New Project, а тип проекта по-прежнему определяется как консольная прикладная программа (Console Application). На рис. 4-13 показано диалоговое окно Edit, которое используется для добавления к проекту различных исходных файлов. Два исходных файла — это CLIENT.C и USERLIB.C.

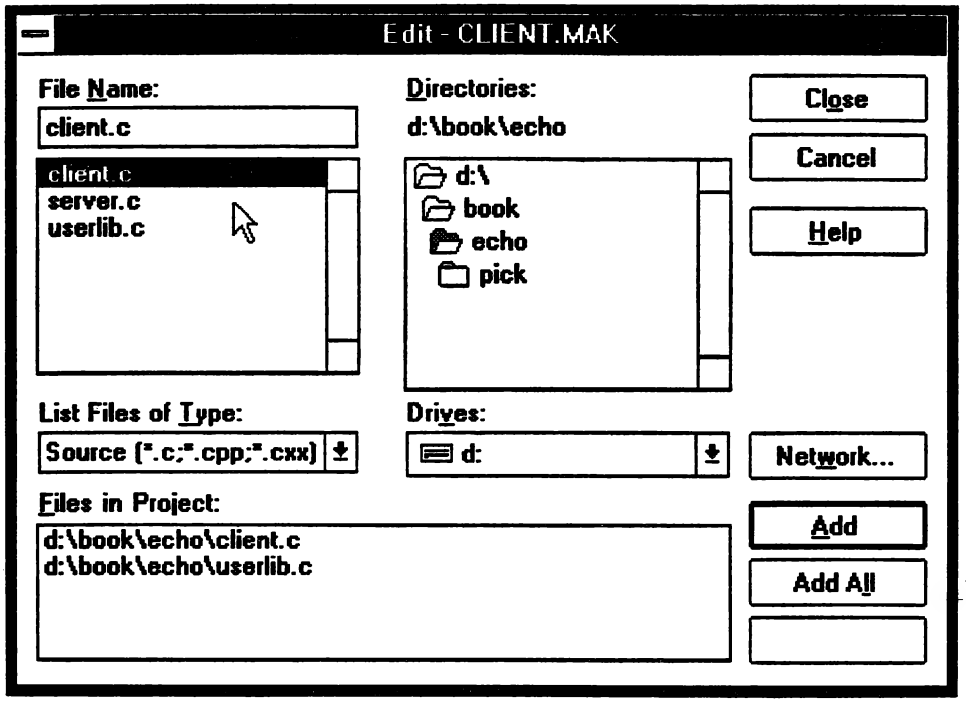


Рис. 4-13. Диалоговое окно Edit.

Построение программы клиента

После создания нового проекта вам необходимо снова установить опции компоновщика, так как это новый проект. Вы добавляете библиотеку WSOK32.LIB к разделу Input окна Linker Options элемента меню Project Options. Установив эту опцию, вы можете осуществить компилирование проекта щелчком мыши по клавише Build. Вы получите картинку на дисплее, аналогичную показанной на рис. 4-14.

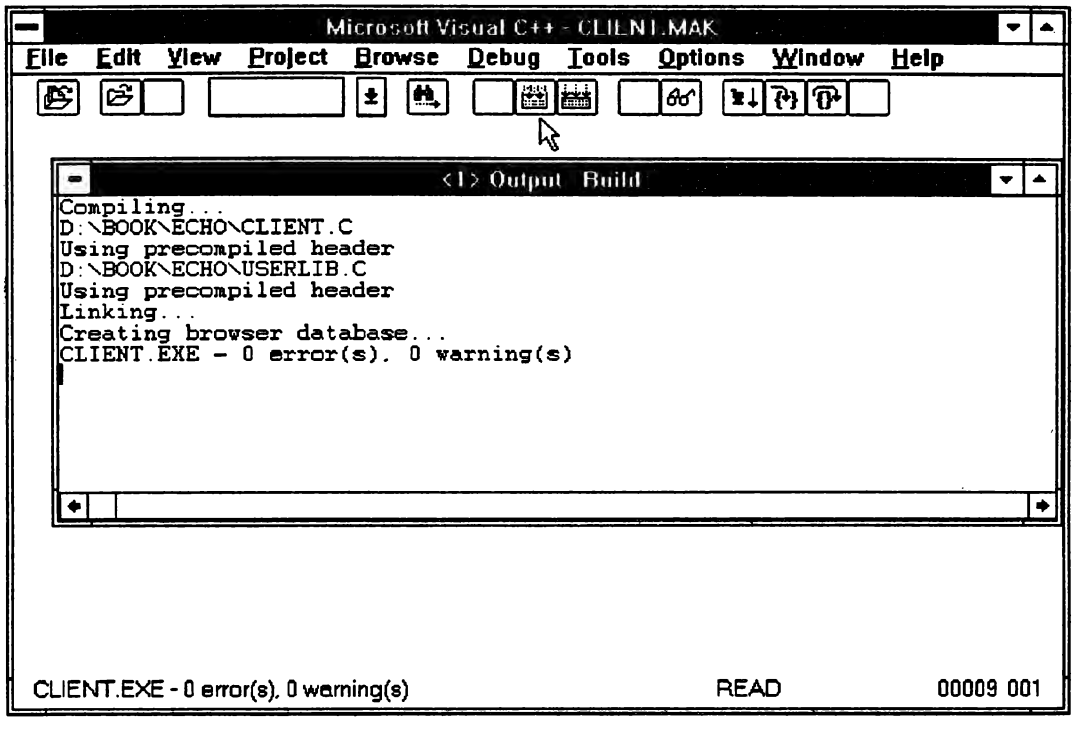


Рис. 4-14. Компилирование проекта.

Программа клиента

Когда вы используете раздел Debug элемента меню Options для установки аргументов для клиента на значение **127.0.0.1 100 1** [current address, port 100, logging enabled] (текущий адрес, порт 100, регистрация задействована), вы можете выполнить программу клиента для получения вида изображения на дисплее, который показан на рис. 4-15.

```
D:\BOOK\ECHO\CLIENT.EXE
Client Echo Program 1.0

        Calling WSAStartup(Version,Details) to locate a Windows DLL
        WSAStartup has returned ImplementationDetails:
          wVersion:      (257)
          wHighVersion:  (257)
          szDescription:'Microsoft Windows Sockets Version 1.1.'
          szSytemStatus:'Running.'
          iMaxSockets:   (32767)
          iMaxUdpDg:     (65467)
          lpVendorInfo   (00000010)


        Calling socket(AddressFormat,Type,Protocol) to Create a Client Socket
        Client Socket (92) has been created

        Calling connect(ServerSocket,AddressPointer,length)
```

Рис. 4-15. Изображение на дисплее после выполнения программы клиента.

Так как программа сервера не выполняется, после нескольких секунд запрос на соединение выключится и клиент выйдет из процесса обработки.

Перенесение программы клиента в DOS

Программа клиента годится для переноса в DOS со внесением незначительных модификаций в исходный код. Если вы копируете исходную программу NT непосредственно в компьютер с MS-DOS и Windows и с установленной PATHWORKS V5.0A для Windows, то вы можете использовать Visual C++ для создания проекта и компилирования программы под MS-DOS и Windows. На рис. 4-16 показаны ошибки, которые могут иметь место, если вы создаете новый проект, включая файлы CLIENT.C и USERLIB.C, и сразу же пытаетесь построить исполнительный файл (executable).

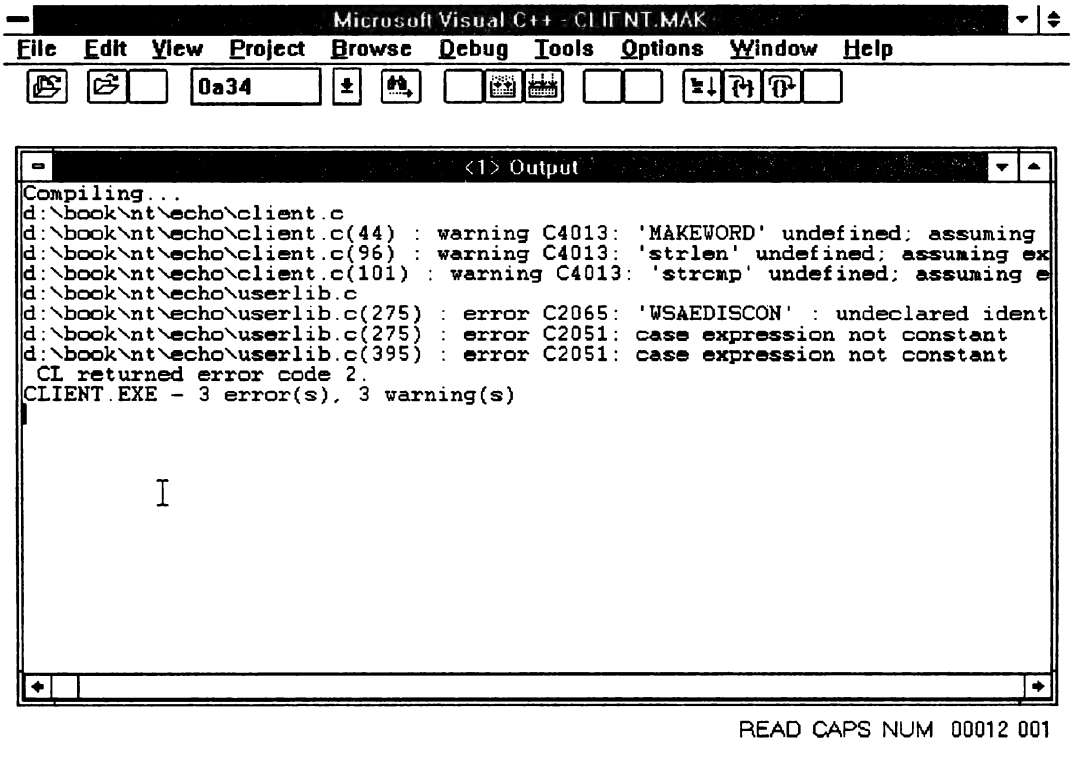


Рис. 4-16. Ошибки при построении исполнительного файла.

Перенесение кода программы

Для того чтобы исправить ошибки, сделанные ранее, внесите следующие изменения в код программы:

В файле USERLIB.H добавьте следующую строку после оператора #endif:

```
#define MAKEWORD(a,b) ((WORD)(((BYTE)(a)):((WORD)((BYTE)(b)))<<8))
```

В файле USERLIB.C, в строке 275, вычеркните ссылку на WSAEDISCON и оператор *break*, который следует за ним. Вам также нужно будет вычеркнуть ссылку, которая теперь будет находиться в строке 393 (ранее она была в строке 395).

Используйте раздел Project элемента меню Options для включения файла WINSOCK.LIB во время фазы компоновщика. Убедитесь, что вы строите проект типа *QuickWin*; в противном случае вы получите сообщение об ошибках (см. рис. 4-17), когда будете строить проект, даже если вы осуществляете компоновку с библиотекой WINSOCK.

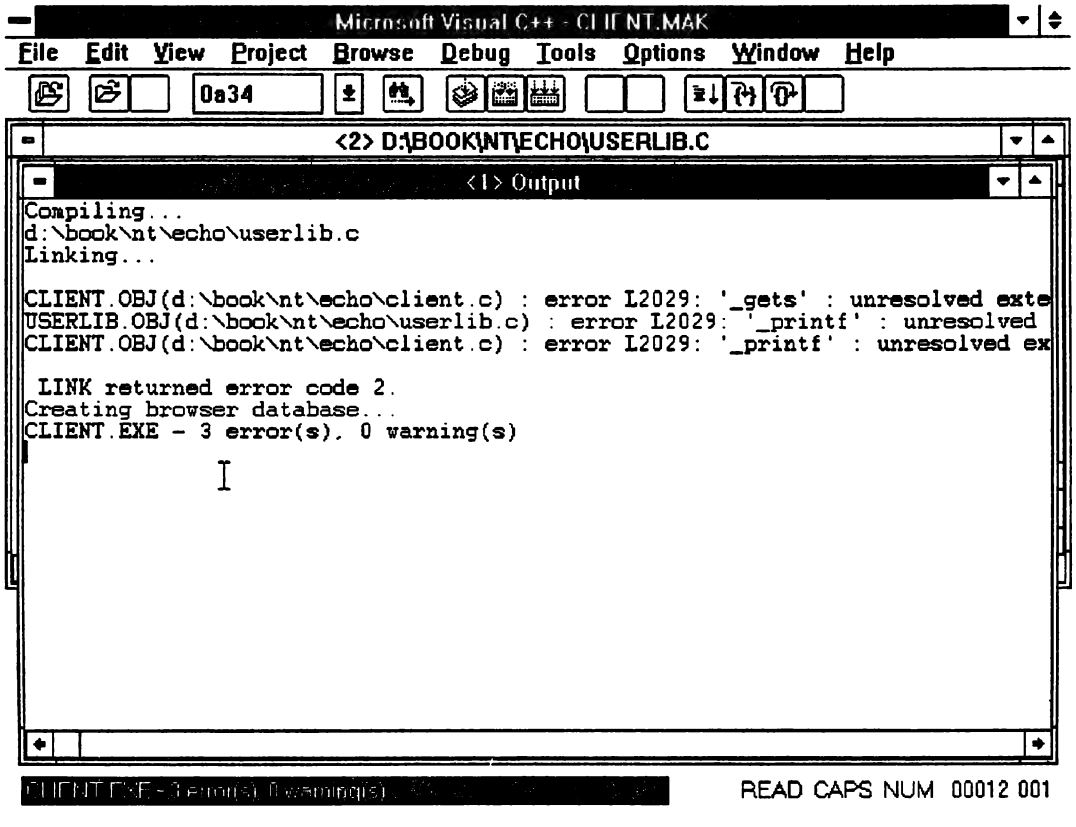


Рис. 4-17. Ошибки при построении.

Построение правильного типа

Под MS-DOS Visual C++ вам необходимо строить прикладную программу типа QuickWin (QuickWin Application), а не прикладную программу Windows. Это позволит вам осуществлять компоновку с нужными стандартными программами ввода/вывода. На рис. 4-18 показано диалоговое окно Project Options с различными типами проектов, приведенными для DOS-версии Visual C++.

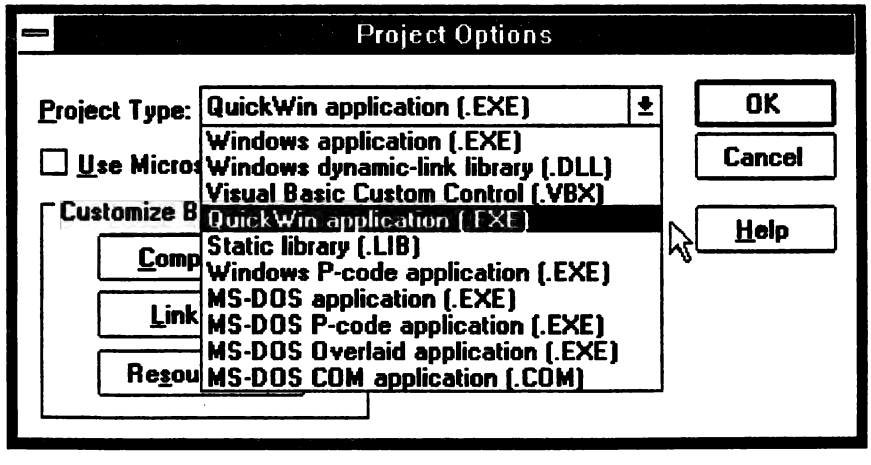


Рис. 4-18. Диалоговое окно Project Options.

Установка каталогов

Когда вы выбрали новый тип проекта, вам следует добавить библиотеку WINSOCK к входным файлам для Linker Options. Вам необходимо добавить этот файл снова, даже если ранее вы уже добавили его к своему проекту, потому что он был потерян, когда вы изменили тип проекта. Вам также следует изменить раздел Directories элемента меню Options для включения файлов PATHWORKS (см. рис. 4-19).

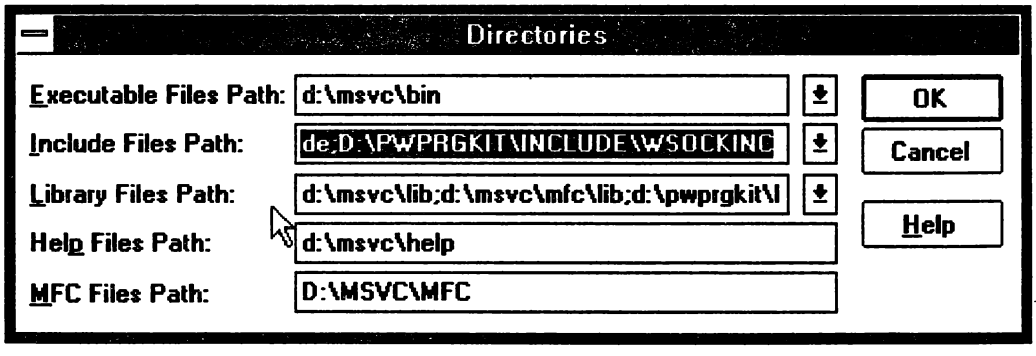


Рис. 4-19. Раздел Directories меню Options.

Маршрут включения должен иметь раздел, указывающий на каталог PWPRGKIT\INCLUDE\WSOCKINC, а маршрут библиотеки должен указывать на каталог PWPRGKIT\LIB. Отметим, что \PWPRGKIT является субкаталогом инсталляции по умолчанию, когда вы инсталлируете PATHWORKS V5.0 Windows/DOS SDK.

Правильное построение DOS

На рис. 4-20 показано успешное построение программы клиента на DOS-машине. Здесь предупреждениями можно пренебречь.

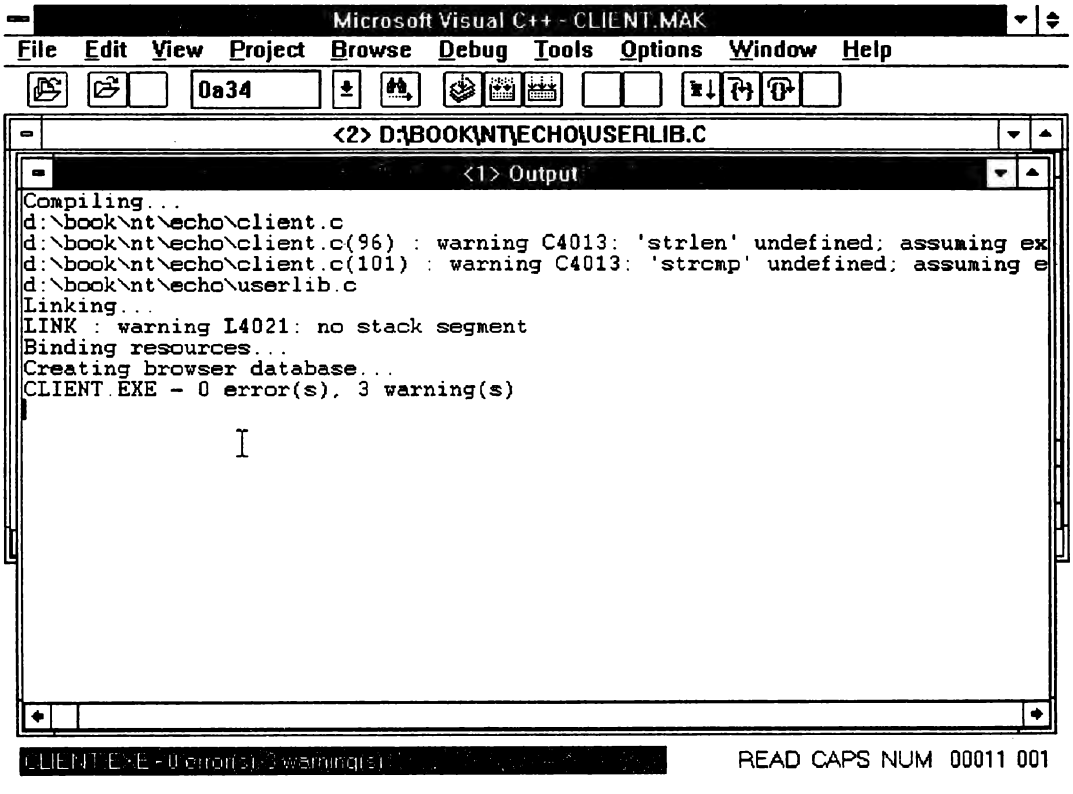


Рис. 4-20. Успешное построение.

Старт диалога

В этом случае сервер стартует на NT-машине с адресом 16.72.32.190, а клиент стартует на той же самой NT-машине.

Ниже дается результат старта сервера без аргументов:

```
D:\BOOK\ECHO>server
 Usage: server port-number [wait-flag] [log-flag]
     if port = 0, then BIND will choose port number
     if wait-flag is 1 [default is 0], then user must hit return to recv
     if log-flag is 1 [default is 0], then detail log messages appear

( D:\BOOK\ECHO>server
   Использование: номер порта сервера [wait-flag] [log-flag]
     если порт = 0, то BIND выберет номер порта;
     если wait-flag = 1 [по умолчанию — 0], то пользователь должен нажать return
              для приема;
     если log-flag = 1 [по умолчанию — 0], то появляется сообщение о деталях
              регистрации; )
```

Этот пример является результатом старта сервера с портом 100, без флага ожидания waitflag и с регистрацией (logging):

```
:\BOOK\ECHO>server 100 0 1
 Server Echo Program 1.0

        Calling WSAStartup(Version,Details) to locate a Windows DLL
        WSAStartup has returned ImplementationDetails:
          wVersion:     (257)
          wHighVersion: (257)
           szDescription:'Microsoft Windows Sockets Version 1.1.'
          szSytemStatus:'Running.'
          iMaxSockets:  (32767)
          iMaxUdpDg:    (65467)
          lpVendorInfo  (00000000)

        Calling socket(AddressFormat,Type,Protocol) to Create a Server Socket
        Server Socket (56) has been created

        Calling bind(Socket,AddressPointer,length) to Bind the socket
        Bind succeeded

        Calling listen(Socket,MaxConnections)
        Successfully changed to a listen socket

        Calling accept(ListenSocket,AddressPointer,length)
        This function will BLOCK until a client is available
```

```
( :\BOOK\ECHO>server 100 0 1
  Server Echo Program 1.0

        Вызов WSAStartup(версия, детали) для определения места DLL
        WSAStartup возвратила подробности реализации:
          wVersion: (257)
          wHighVersion: (257)
          szDescription: 'Microsoft Windows Sockets Version 1.1
          szSystemStatus: 'Running'
          iMaxSockets: (32767)
          iMaxUdpDg: (65467)
          lpVendorInfo (00000000)

        Вызов гнезда (AddressFormat,Type,Protocol) для создания гнезда сервера
        Гнездо Сервера (56) создано

        Вызов bind(Socket,AddressPointer,length) для привязывания гнезда
        Привязывание выполнено успешно

        Вызов listen(Socket,MaxConnections)
        Успешное изменение в гнезде прослушивания

        Вызов accept(ListenSocket,AddressPointer,length)
        Эта функция будет БЛОКИРОВАТЬСЯ, пока клиент не будет доступен  )
```

Здесь сервер ждет старта клиента. Первый клиент стартует и посылает следующие сообщения:

```
hello
testing
slog
test
message
slog
bye
```

Когда клиент посылает команду *bye*, то после этого он выходит из работы.

Когда посылаются команды клиента, сервер отвечает следующим образом:

```
accept succeeded: A client is available

Calling recv(Socket,buffer,SizeOfBuffer,Flags)
socket:036>[006]hello\000                                hello
Calling send(Socket,buffer,SizeOfBuffer,Flags)
6 characters sent to the Client
```

```
        Data: ifmmp
        Calling recv(Socket,buffer,SizeOfBuffer,Flags)
 socket:036>[008]testing\000                                testing
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        8 characters sent to the Client
        Data: uftujoh
        Calling recv(Socket,buffer,SizeOfBuffer,Flags)
 socket:036>[005]slog\000                                   slog
 Client sent slog command. LogFlag toggled
 socket:036>[005]test\000                                   test
 socket:036>[008]message\000                                message
 socket:036>[005]slog\000                                   slog
 Client sent slog command. LogFlag toggled
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        5 characters sent to the Client
        Data: tmph
        Calling recv(Socket,buffer,SizeOfBuffer,Flags)
 socket:036>[004]bye\000                                    bye
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        4 characters sent to the Client
        Data: czf
        Calling recv(Socket,buffer,SizeOfBuffer,Flags)
 Connection Reset, Listening for a New Client..
        Calling accept(ListenSocket,AddressPointer,length)
        This function will BLOCK until a client is available

(
        Вызов прошел: клиент доступен
        Вызов recv(Socket,buffer,SizeOfBuffer,Flags)
socket:036>[006]hello\000                                   hello
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        6 знаков посланы клиенту
        Данные:ifmmp
        Вызов recv(Socket,buffer,SizeOfBuffer,Flags)
socket:036>[008]testing\000                                 testing
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        8 знаков посланы клиенту
        Данные: uftujoh
        Вызов recv(Socket,buffer,SizeOfBuffer,Flags)
socket:036>[005]slog\000                                    slog
Клиент послал команду slog. LogFlag переброшен
socket:036>[005]test\000                                    test
socket:036>[008]message\000                                 message
socket:036>[005]slog\000                                    slog
Клиент послал команду slog. LogFlag переброшен
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        5 знаков посланы клиенту
        Данные:tmph
        Вызов recv(Socket,buffer,SizeOfBuffer,Flags)
socket:036>[004]bye\000                                     bye
```

```
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        4 знака посланы клиенту
        Данные:czf
        Вызов recv(Socket,buffer,SizeOfBuffer,Flags)
Сброс соединения, прослушивание нового клиента...
        Вызов accept(ListenSocket,AddressPointer,length)
        Эта функция будет БЛОКИРОВАТЬСЯ, пока клиент не будет доступен
```

Новый клиент стартует и посылает следующие сообщения:

```
new client
clog
hello
clog
quit
```

Сервер отвечает следующим:

```
        accept succeeded: A client is available

        Calling recv(Socket,buffer,SizeOfBuffer,Flags)
 socket:036>[011]new client\000                          new client
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        11 characters sent to the Client
        Data: ofx!dmjfou
        Calling recv(Socket,buffer,SizeOfBuffer,Flags)
 socket:036>[005]clog\000                                clog
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        5 characters sent to the Client
        Data: dmph
        Calling recv(Socket,buffer,SizeOfBuffer,Flags)
 socket:036>[006]hello\000                               hello
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        6 characters sent to the Client
        Data: ifmmp
        Calling recv(Socket,buffer,SizeOfBuffer,Flags)
 socket:036>[005]clog\000                                clog
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        5 characters sent to the Client
        Data: dmph
        Calling recv(Socket,buffer,SizeOfBuffer,Flags)
 socket:036>[005]quit\000                                quit
 Client sent quit command, Server exiting...
 Exiting the Server ....

(       Прием успешен: клиент доступен
        Вызов recv(Socket,buffer,SizeOfBuffer,Flags)
socket:036>[011]новый клиент\000 new client
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        11 знаков посланы клиенту
```

```
        Данные: ofx!dmjfou
        Вызов recv(Socket,buffer,SizeOfBuffer<Flags)
socket:036>[005]clog\000                                    clog
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        5 знаков посланы клиенту
        Данные:dmph
        Вызов recv(Socket,buffer,SizeOfBuffer,Flags)
socket:036>[006]hello\000                                   hello
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        6 знаков отосланы клиенту
        Данные:ifmmp
        Вызов recv(Socket,buffer,SizeOfBuffer,Flags)
socket:036>[005]clog\000                                    clog
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        5 знаков отосланы клиенту
        Данные: dmph
        Вызов recv(Socket,buffer,SizeOfBuffer,Flags)
socket:036>[005]quit\000                                    quit
Клиент послал команду quit, Сервер выходит из обработки...
Выход сервера из обработки...          )
```

Вот пример старта клиента без аргументов:

```
 D:\BOOK\ECHO>client
  usage: client inet-address port [Log-flag]
    Log-flag of 1 [0 is default] means show log details .

(   D:\BOOK\ECHO>client
      usage: client inet-address port [LogFlag]
       Log-flag равный 1 [по умолчанию — 0] означает показ деталей регистрации   )
```

Вот вывод от первого клиента, который подключился к серверу:

```
 D:\BOOK\ECHO>client 127.0.0.1 100 1
 Client Echo Program 1.0

        Calling WSAStartup(Version,Details) to locate a Windows DLL
        WSAStartup has returned ImplementationDetails:
          wVersion:     (257)
          wHighVersion: (257)
           szDescription:'Microsoft Windows Sockets Version 1.1.'
          szSytemStatus:'Running.'

          iMaxSockets:  (32767)
          iMaxUdpDg:    (65467)
          lpVendorInfo  (00000010)

        Calling socket(AddressFormat,Type,Protocol) to Create a Client Socket
        Client Socket (56) has been created
```

```
        Calling connect(ServerSocket,AddressPointer,length)
        Connect succeeded: A Server is available

Enter data to transmit to the server
Client>hello                                          hello
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        6 characters sent to the Server
        Data: hello
socket:056>[006]ifmmp\000
Client>testing                                        testing
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        8 characters sent to the Server
        Data: testing
socket:056>[008]uftujoh\000
Client>slog                                           slog
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        5 characters sent to the Server
        Data: slog
socket:056>[005]tmph\000
Client>test                                           test
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        5 characters sent to the Server
        Data: test
socket:056>[005]uftu\000
Client>message                                        message
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        8 characters sent to the Server
        Data: message
socket:056>[008]nfttbhf\000
Client>slog                                           slog
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        5 characters sent to the Server
        Data: slog
socket:056>[005]tmph\000
Client>bye                                            bye
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        4 characters sent to the Server

        Data: bye
Client closing socket
press Return to Exit Client
<Return>

( D:BOOK\ECHO>client 127.0.0.1 100 1
Client Echo Program 1.0
        Вызов WSAStartup(Version,Details) для определения места Windows DLL
        WSAStartup возвратил ImplementationDetails (детали реализации):
          wVersion: (257)
          wHighVersion: (257)
          szDescription: 'Microsoft Windows Sockets Version 1.1'
```

```
szSystemStstus: 'Running'
iMaxSockets: (32767)
iMaxUdpDg: (65467)
lpVendorInfo (00000010)

Вызов гнезда(AddressFormat,Type,Protocol) для создания гнезда клиента
Гнездо клиента (56) создано

Вызов connect(ServerSocket,AddressPointer,length)
Соединение осуществлено успешно: сервер доступен

Ввод данных для передачи на сервер
Client>hello                                        hello
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        6 знаков посланы на сервер
        Данные:hello
Socket:056>[006]ifmmp\000
Client>testing                                      testing
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        8 знаков посланы на сервер
        Данные:testing socket:056>[008]uftujoh\000
Client> slog                                        slog
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        5 знаков посланы на сервер
        Данные: slog socket:056>[005]tmph\000
Client>test                                         test
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        5 знаков посланы на сервер
        Данные: test socket:056>[005]uftu\000
Client>message                                      message
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        8 знаков посланы на сервер
        Данные: message
socket:056>[008]nfttbhf\000
Client>slog                                         slog
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        5 знаков посланы на сервер
        Данные: slog socket:056>[005]tmph\000
Client>bye                                          bye
        Вызов send(Socket,buffer,SizeOfBuffer,Flags)
        4 знака посланы на сервер

        Данные: bye
Клиент закрывает гнездо
нажмите Return для выхода клиента из обработки
<Return>
```

Это второй клиент, который подключился к серверу:

```
D:\BOOK\ECHO>client 127.0.0.1 100 1
```

```
Client Echo Program 1.0

        Calling WSAStartup(Version,Details) to locate a Windows DLL
        WSAStartup has returned ImplementationDetails:
          wVersion:     (257)
          wHighVersion: (257)
           szDescription:'Microsoft Windows Sockets Version 1.1.'
          szSytemStatus:'Running.'
          iMaxSockets:  (32767)
          iMaxUdpDg:    (65467)
          lpVendorInfo  (00000010)

        Calling socket(AddressFormat,Type,Protocol) to Create a Client Socket
        Client Socket (56) has been created

        Calling connect(ServerSocket,AddressPointer,length)
        Connect succeeded: A Server is available

Enter data to transmit to the server
Client>new client                                        new client
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        11 characters sent to the Server
        Data: new client
socket:056>[011]ofx!dmjfou\000
Client>clog                                              clog

        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        5 characters sent to the Server
        Data: clog
Client noticed clog command. LogFlag toggled
socket:056>[005]dmph\000
Client>hello                                             hello
socket:056>[006]ifmmp\000
Client>clog                                              clog
Client noticed clog command. LogFlag toggled
socket:056>[005]dmph\000
Client>quit                                              quit
        Calling send(Socket,buffer,SizeOfBuffer,Flags)
        5 characters sent to the Server
        Data: quit
Client sending quit command, Client exiting...

( D:BOOK\ECHO>client 127.0.0.1 100 1
Client Echo Program 1.0
        Вызов WSAStartup(Version,Details) для определения места Windows DLL
        WSAStartup возвратил ImplementationDetails:
          wVersion: (257) wHighVersion: (257)
          szDescription: 'Microsoft Windows Sockets Version 1.1'
          szSystemStatus: 'Running' iMaxSockets: (32767)
          iMaxUdpDg: (65467)
```

```
        lpVendorInfo: (00000010)

    Вызов гнезда(AddressFormat,Type,Protocol) для создания гнезда клиента
    Гнездо клиента (56) создано

    Вызов connect(ServerSocket,AddressPointer,length)
    Соединение осуществлено успешно: сервер доступен

Ввод данных для передачи на сервер
Client>new client                                   new client
    Вызов send(Socket,buffer,SizeOfBuffer,Flags)
    11 знаков посланы на сервер
    Данные: новый клиент
socket:056>[011]ofx!dmjfou\000
Client>clog                                         clog
    Вызов send(Socket,buffer,SizeOfBuffer,Flags)
    5 знаков посланы на сервер
    Данные: clog
Клиент обнаружил команду clog. LogFlag перебрасывается
socket:056>[005]dmph\000
Client>hello                                        hello
socket:056>[006]ifmmp\000
Client>clog                                         clog
Клиент обнаружил команду clog. LogFlag перебрасывается
socket:056>[005]dmph\000
Client>quit                                         quit
    Вызов send(Socket,buffer,SizeOfBuffer,Flags)
    5 знаков посланы на сервер
    Данные: quit
Клиент посылает команду quit, клиент выходит из обработки...
```

Глава 5

Усовершенствованное программирование гнезд: мультирежим системы "клиент-сервер"

ВВЕДЕНИЕ

Назначение

Основной целью настоящего примера программы является разрешение обслуживания одновременно нескольких клиентов. Вводятся новый клиент и сервер на основании работы клиента и сервера в режиме "эхо", о которой было сказано в предыдущей главе. Клиент имеет дополнительные характеристики для поддержки асинхронного, или неблокирующего, режима работы.

Новый клиент

Программа клиента требует два аргумента командной строки и принимает еще два:

- *inet-address* — Этот аргумент представляет собой межсетевой адрес сервера, выраженный в виде 4-байтовой последовательности, например 16.72.32.185.
- *port* — Этот аргумент представляет собой адрес порта, к которому привязан сервер на конкретной машине, на которой выполняется программа сервера.
- *[log-flag]* — Этот опциональный параметр определяет первоначальное состояние глобальной переменной *LogFlag*. Значение 1 означает показ деталей регистрации. Если значение не определено, то будет использоваться величина по умолчанию. Это — 0 [нет регистрации].
- *[sync-flag]* — Этот опциональный параметр определяет первоначальное состояние глобальной переменной *SyncFlag*. Значение этого параметра, равное 1, означает установку асинхронного режима обработки (Asynchronous Mode). Если этот параметр не определен, то будет использоваться величина по умолчанию. Это — 0, т.е. синхронный (блокирующий) режим работы.

Программа клиента осуществляет следующие шаги:

1. Инициализирует программу.
2. Инициализирует сеть.
3. Создает оконечный пункт.

4. Осуществляет подключение к другому оконечному пункту (установленному сервером).
5. Осуществляет подсказку относительно местных данных от пользователя.
6. Устанавливает синхронный режим для гнезда.
7. Посылает данные на сервер.
8. Обрабатывает любые команды в местных данных:

exit — Local: client! Exit the client; Remote: server! Client is exiting
Местная: клиент! Выход клиента.Удаленная: сервер! Клиент выходит из работы Соответствует нормальному выходу клиента из работы

quit — Local: client! Exit the client; Remote: server! Exit the server Corresponds to normal shutdown
Местная: клиент! Выход клиента; Удаленная: сервер! Выход сервера Соответствует нормальному окончанию работы

bye — Local:client! Exit the client; Remote: no bye command corresponds to abnormal client exit
Местная: клиент! Выход клиента; Удаленная: аварийному выходу клиента из работы не ставится в соответствие никакой команды bye

clog — Local: client! Toggle the client Log flag
Местная: клиент! Перебрасывает флаг клиента Log flag

csync — Local: client! Toggle the client Sync flag
Местная: клиент! Перебрасывает флаг клиента Sync flag

coob — Local: client! Toggle the client Send flag
Местная: клиент! Перебрасывает флаг клиента Send flag

clots — Local: client! Send 4000 bytes of data to the server
Местная: клиент! Посылает 4000 байт данных на сервер

Отметим, что этот клиент не получает данных от сервера. Он не будет работать с предыдущим сервером, потому что тот сервер будет пытаться послать данные этому клиенту и в конечном счете заполнит входной буфер клиента и выходной буфер сервера, после чего сервер будет заблокирован.

Новый сервер

Программа сервера требует одного аргумента командной строки и принимает еще три опциональных параметра:

- Port-number — Это номер порта, который будет отведен для сервера. Если вы даете значение 0, тогда функция *bind* выберет для вас неиспользуемый порт.
- [wait-flag] — Этот опциональный аргумент используется для установки начального состояния для переменной *WaitFlag*. Если вы не даете этот аргумент, тогда будет использовано значение по умолчанию, равное 0. Если значение *waitflag* равно 1, то вам выдается на сервере подсказка нажать Return

до того, как данные будут получены от клиента. Это позволяет вам имитировать состояние занятости сервера, когда клиент пытается послать данные.

- *[log-flag]* — Этот опциональный аргумент используется для установки начального состояния для переменной LogFlag. Если вы не поставляете этот аргумент, то будет использовано значение по умолчанию, равное 0. Значение, равное 1, означает показ деталей регистрации (log details).
- *[process-flag]* — Этот опциональный аргумент используется для установки начального состояния для переменной *ProcessAllExceptions*. Если вы не поставляете этот аргумент, то будет использовано значение по умолчанию, равное 1. Значение, равное 1, означает обработку всех исключительных (внедиапазонных) сообщений — Out of Band (OOB) — до обработки каких-либо нормальных сообщений.

Сервер выполняет следующие шаги:

1. Инициализирует программу.
2. Инициализирует сеть.
3. Создает оконечный пункт.
4. Регистрирует оконечный пункт с сетью и вставляет его в массив гнезд.
5. Ждет проявления активности на любом гнезде из контролируемого массива гнезд.
6. Если активность имеет место на гнезде оконечного пункта, тогда принимает клиента, пытающегося осуществить связь, и добавляет это гнездо в массив гнезд.
7. Обрабатывает все гнезда, которые имеют исключительные данные, доступные для сервера.
8. Если установлен флаг *ProcessAllExceptions* и исключительные данные только что были обработаны, тогда возвращается к шагу 5 для обработки каких-либо дополнительных исключительных данных.
9. Для всех гнезд с нормальной активностью:
   a. Проверяет на наличие сетевых ошибок:
      1) Connection Reset — сброс соединения;
      2) Zero Length Data — данные нулевой длины.

   b. Обрабатывает удаленные [Remote] команды, посланные от клиента:
      1) *Exit* — переход к шагу 5, ожидание другого клиента;
      2) *Quit* — вывод сервера из работы;
      3) *slog* — переброс флага сервера log flag;
      4) *swait* — переброс флага сервера wait flag;
      5) *sall* — переброс флага ProcessAllExceptions.
10. Переходит к шагу 5 для ожидания новой активности на любом из контролируемых гнезд.

РАБОТА С КЛИЕНТАМИ В МУЛЬТИРЕЖИМЕ (С ПОЯСНЕНИЯМИ)

```
#include <winsock.h>
#include <stdio.h>
#include <stdlib.h>
#include "Userlib.h"

char * NotMe = "Server";
char * Me    = "Client";
int  LogFlag;
int  SyncFlag;
int  SendFlag=0;
```

WINSOCK.H

Эта клиентская программа создается как консольная программа Windows NT. Для ее изменения у клиента MS-DOS вам следует заменить файл включения WINSOCK.H на PRGPRE.H. Помните, что PRGPRE.H подходит для PATHWORKS фирмы DEC. Если у вас имеется библиотека гнезд другого поставщика, вам следует включить соответствующий файл этого поставщика. Если вы создаете прикладную программу QuickWin версии 3.1 Windows, то вам следует сохранить файл включения WINSOCK.H.

STDx.H

Эти файлы включаются, чтобы осуществить доступ к стандартным функциям библиотеки С, используемым для ввода и вывода в этой программе.

UserLib.h

Это специальный библиотечный файл, используемый в примерах для "завертывания" стандартных вызовов гнезд и переадресации их к функциям *User_x*. Функции *User_x* имеют встроенную логику для вывода на печать различных сообщений, если установлен флаг *LogFlag*.

Глобальные переменные

Для идентификации того, кто именно посылает сообщения (клиент или сервер), в UserLib используются две глобальные переменные — *Me* и *NotMe*. Более подробно об этих переменных смотрите пример с клиентом ECHO.

Переменная *LogFlag* используется в функциях UserLib в качестве условия перед операторами *printf*.

Переменная *SyncFlag* используется для определения того, будет ли отсылка данных осуществляться в синхронном или асинхронном режиме.

Переменная *Sendflag* используется для определения того, относится ли посылаемое сообщение к нормальному сообщению или же к внедиапазонному сообщению (Out-of-Band message).

```
_cdecl main(int argc, char *argv[])
  {
    int rv;
    WORD        VersionRequested;
    WSADATA     ImplementationDetails;
    SOCKET      ClientSocket;
    SOCKADDR_IN ServerSocketAddress;

    if ((argc <=2) || (argc >5))
     { printf(" usage: %s inet-address port [Log-flag] [Sync- flag]\n",argv[0]);
       printf(" Log-flag of 1 [0 is default] means show log details\n");
       printf(" Sync-flag of 1 [0 is default] means Asynchronous behavior\n");
       return(0);
     }
    if (4<=argc) LogFlag = atoi(argv[3]);
    else LogFlag = 0;
    if (5==argc) SyncFlag = atoi(argv[4]);
    else SyncFlag = 0;
```

Инициализация программы

Первая часть процедуры *main* используется для объявления локальных (местных) переменных и для считывания аргументов командной строки. Если команда вызывается с неправильным количеством аргументов, тогда на печать выводится сообщение об использовании и программа выходит из работы.

Если представлено правильное количество аргументов, тогда аргументы считываются в соответствующие переменные. В противном случае, если опциональные аргументы не представлены, используются значения по умолчанию.

Когда вы используете библиотеку WINSOCK, которая является библиотечным файлом, используемым в этом примере, требуются только две местные переменные — *VersionRequested* и *ImplementationDetails*. Если вы используете MS-DOS, эти две переменные вам не нужны. Значения будут помещены в эти переменные во время фазы инициализации сети.

Для удержания оконечного пункта, который подлежит использованию клиентом для осуществления связи с сетью, используется переменная *ClientSocket*. *ServerSocketAddress* удержит адрес *ServerSocket*, с которой клиент соединяется. Он заполняется клиентом до вызова функции *connect*.

```
printf("Client Echo Program 1.0\n\n");
VersionRequested = MAKEWORD(1,1);
rv = WSAStartup (VersionRequested, &ImplementationDetails);
if (0 != rv ) Error ();
```

Инициализация сети

Весь этот раздел можно опустить, если вы переносите эту программу на MS-DOS и не используете библиотеку WINSOCK. Библиотечный PWSOCK.LIB операционной системы PATHWORK фирмы DEC не требует инициализации сети.

Так как этот пример программы выполняется на Windows NT и компонуется с библиотекой WINSOCK, сеть должна быть инициализирована вызовом *WSAStartup*. Это требует версии 1.1 Sockets DLL и просит, чтобы детали реализации сети были перенесены обратно в структуру *ImplementationDetails*. Далее следует та же процедура, если вы прогоняете эту программу на Windows Version 1.3 и используете WINSOCK.

Файл UserLib.h заново определяет *WSAStartup* как *User_WSAStartup*, поэтому в этой программе вы можете задействовать регистрацию.

Если *WSAStartup* возвращает ненулевое значение, указывающее, что возникла ошибка, вызывается функция *ErrorStop()*. Функция *ErrorStop* выведет на печать индикатор к сообщению об ошибке и затем прекратит выполнение программы.

```
 ClientSocket = socket (PF_INET,SOCK_STREAM,0);
 if (INVALID_SOCKET == ClientSocket) ErrorStop();
 ServerSocketAddress.sin_family          = AF_INET;
 ServerSocketAddress.sin_addr.S_un.S_addr    = inet_addr(argv[1]);
 ServerSocketAddress.sin_port            = atoi(argv[2]);

 rv = connect (ClientSocket,(LPSOCKADDR)&ServerSocketAddress, sizeof(ServerSocketAddress));
 if (0 != rv) ErrorStop();

 printf("Enter data to transmit to the server\n");
 UserGetData(ClientSocket);

return(0);
}
```

Создание оконечного пункта и регистрация его с сетью

Функция *socket()* используется для создания оконечного пункта. Три параметра, которые использует *socket()*, указывают, что гнездо использует межсетевой (Internet) протокол, является поточным гнездом и не имеет

никаких специальных атрибутов. Если функция возвращает -1, которая является значением INVALID_SOCKET (недействительное гнездо), тогда вы вызываете *ErrorStop()* и прекращаете выполнение программы.

Допустим, что гнездо было успешно создано, и теперь вы можете соединить гнездо с существующим сервером, размещенным в сети, вызовом функции *connect()*. Вы можете привязать гнездо к адресу, если вы собираетесь выполнять несоединенные сетевые функции без логического соединения. В этой программе цель заключается в том, чтобы подсоединиться к серверу, поэтому вы вызываете *connect()*.

Функция *connect()* должна знать сетевой адрес сервера. Это осуществляется определением компонент специальной структуры и передачей указателя структуры в *connect()*. Структура, на которую указывает *connect()*, должна быть структурой SOCKADDR, которая не зависит от протокола. Структура, которую вы заполняете, является структурой SOCKADDR_IN для межсетевого (Internet) варианта структуры SOCKADDR. Более подробную информацию смотрите в предыдущей главе (клиент в режиме "эхо").

После того как соединение осуществлено, вы вызываете функцию *UserGetData* для обработки данных от пользователя. Когда эта функция заканчивается, тогда осуществляется выход клиента из обработки.

```
void UserGetData(SOCKET ClientSocket)
 { long arg;
   char buffer[5000];
   int rv;
   while(1==1
     {
     printf("Client>");
     gets(buffer);
     arg = SyncFlag;
     rv = ioctlsocket (ClientSocket,FIONBIO,&arg);
     rv = send (ClientSocket,buffer,strlen(buffer)+1,SendFlag);
     if (SOCKET_ERROR == rv) ErrorStop();
```

Обработка данных пользователя

Для того чтобы данные создавались или считывались от пользователя и посылались на сервер, в локальном стеке предусмотрен буфер на 5000 байт. Затем программа вводится в бесконечный цикл *[while (1 = = 1)]*, до тех пор пока не будет введена команда пользователя, которая заставляет выполняться <Return>.
Функция *ioctlsocket()* используется для установки атрибутов гнезда.

Следующие команды могут быть переданы в функцию в качестве второго параметра, в то время как третьим параметром является адрес переменной *arg*:

- FIONBIO — Устанавливает неблокирующий режим, если *arg* содержит не нулевую величину.
- FIONREAD — Возвращает счет доступных в сети байт в *arg*.

Функция *send* используется для передачи данных на сервер. Отметим, что главное различие между этой *send* и *send* у клиента в режиме "эхо" (ECHO client) заключается в четвертом параметре. Этот параметр определяет, как передавать данные:

- 0 — передача данных в нормальном диапазоне;
- MSG_OOB — Передача данных как исключение, т.е. во внедиапазонном режиме (OOB).

Отметим, что некоторые реализации имеют пределы относительно того, сколько данных OOB DATA можно отослать.

Если во время выполнения функции *send* возникает ошибка, то вызывается функция *ErrorStop()* и программа выходит из выполнения.

```
if (0==strcmp("exit",buffer))
    { printf("Client sending exit command,Client exiting...\n");
    return;
    }
if (0==strcmp("quit",buffer))
    { printf("Client sending quit command, Client exiting...\n");
        return;
    }
if (0==strcmp("bye",buffer))
    { printf("Client closing socket\n");
     rv =closesocket(ClientSocket);
     if (SOCKET_ERROR == rv) Error ();
     printf("press Return to Exit Client\n");
     getchar();
     return;
    }
```

Завершение работы клиента

После отсылки данных на сервер данные проверяются на предмет определения того, не ввел ли пользователь одну из следующих команд завершения работы:

- *exit* — клиент прекращает работу, сервер закрывает гнездо и убирает его из массива контролируемых гнезд;

- *quit* — клиент прекращает работу, сервер прекращает работу;
- *bye* — клиент закрывает свое гнездо, не оповещая об этом сервер. Используется для эмуляции сетевого "всплеска" ("burp"), когда соединение теряется.

```
if (0==strcmp("clog",buffer))
    { LogFlag = (1==LogFlag) ? 0: 1;
      printf("Client noticed clog command. LogFlag now %d\n",LogFlag);
    }
if (0==strcmp("csync",buffer))
    { SyncFlag = (1==SyncFlag) ? 0: 1;
      printf("Client noticed csync command. SyncFlag now %d\n",SyncFlag);
    }
if (0==strcmp("coob",buffer))
    { SendFlag = (0==SendFlag)?MSG_OOB : 0;
      printf("Client noticed coob command. SendFlag now %d\n",SendFlag);
    }
```

Переключение местных переменных

Если команда прекращения работы (termination command) не используется, тогда вы осуществляете проверку на предмет того, не была ли указана одна из следующих команд:

- clog — перебрасывает флаг log flag. Включает и выключает сообщения регистрации;
- csync — перебрасывает флаг sync flag. Включает и выключает блокирующий режим;
- coob — перебрасывает флаг send flag. Включает и выключает передачу исключительных сообщений.

Когда сервер получает данные, он проверяет их на предмет наличия следующих команд переброса:

- slog — перебрасывает флаг сервера *LogFlag*;
- swait — перебрасывает флаг сервера *WaitFlag*;
- sall — перебрасывает флаг сервера *ProcessAllExceptions*.

```
if (0==strcmp("clots",buffer))
   { static char charToSend='a';
     int i;
     printf("Client noticed clots command. 4000 '%c's send\n");
     for (i=0;i<4000;i++) buffer[i]=charToSend;
     rv = send (ClientSocket,buffer,strlen(buffer)+1,0);
     if (SOCKET_ERROR == rv) ErrorStop ();
     charToSend++;
```

```
        if ('z'<charToSend) charToSend='a';
    }
  }
}
```

Отсылка больших объемов данных

Команда *clots* используется для отсылки к серверу 4000 байт данных. Эта команда используется для заполнения входного буфера сервера и затем заполнения выходного буфера клиента. Когда оба буфера будут заполнены, следующая отсылка либо:

- блокируется, если гнездо находится в блокирующем режиме. Отсылка данных не осуществляется, пока сервер не выдаст команду *recv* для получения данных из своего входного буфера. Это освобождает достаточно пространства, для того чтобы функция *send* клиента разблокировалась;
- возвращает ошибку WSAEWOULDBLOCK, если гнездо находится в неблокирующем режиме. Данные не передаются, так как передача данных вызвала бы блокировку функции *send*. При необходимости клиент имеет возможность выполнять другие функции.

Логика этой команды передаст, когда она вызвана в первый раз, 4000 a знаков, в следующий раз — 4000 b знаков, и так далее каждый раз заменяя букву алфавита на следующую, пока она не достигнет буквы z. Когда передана буква z, эта команда стартует опять с буквой a.

РАБОТА С СЕРВЕРАМИ В МУЛЬТИРЕЖИМЕ (С ПОЯСНЕНИЯМИ)

```
#include <winsock.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "Userlib.h"

char * Me = "Server";
char * NotMe = "Client";
int    WaitFlag;
int    LogFlag;
int    ReadFlag=0;
int    ProcessAllExceptions=1;
```

Заголовочные файлы

Для клиента и для сервера используются одни и те же заголовочные файлы, с тем лишь исключением, что сервер имеет дополнительный

стандартный файл С, так как он использует функции, определенные в *ctype.h*. В этом примере используется заголовочный файл WINSOCK.h, так как файл создается как консольная программа Windows NT или как прикладная программа QuickWin Windows 3.1. Если эта программа переносится на MS-DOS, тогда может потребоваться другой заголовочный файл в зависимости от стека поставщика.

Глобальные переменные

Отметим, что две переменные *Me* и *NotMe* имеют у клиента обратные значения соответственно. Этим достигается правильная работа функций в UserLib.

Сервер имеет четыре флаговых переменных:

- *WaitFlag* — используется для принятия решения, нужно ли ждать перед обработкой данных клиента;
- *LogFlag* — используется для принятия решения, следует ли выводить на печать сообщения регистрации;
- *ReadFlag* — используется для определения того, как считывать данные от клиента — во внедиапазонном (OOB) режиме или нормальном;
- *ProcessAllExceptions* — используется для определения того, все ли данные OOB нужно считывать.

```
_cdecl main(int argc, char *argv[])
 {
 char    buffer[100];
 long    lrv;
 int     rv;
 int     Server Stop Flag;
 unsigned short port;

 WORD          VersionRequested;
 WSADATA       ImplementationDetails;
 SOCKET        ServerSocket;
 SOCKET        ClientSocket;
 SOCKADDR_IN   ServerSocketAddress;
 SOCKADDR_IN   ClientSocketAddress;
 fd_set        AllSockets;
 fd_set        ReadSockets;
 fd_set        ExceptSockets;
```

Объявление местных (локальных) переменных

Сервер объявляет несколько местных переменных, которые используются для различных целей. Буфер массива знаков используется для приглашения пользователю нажать <Return>. Переменные *lrv* и *rv*

используются для хранения значений возврата; *lrv* — для хранения больших значений, а *rv* — для хранения целых чисел. *ServerStopFlag* используется для принятия решения о выходе сервера из работы. *Port* используется для указания того, к какому порту будет присоединяться сервер.

Сервер, как и клиент, должен осуществить инициализацию сети, когда программа сервера выполняется под библиотекой WINSOCK. Следующие две из объявленных переменных — *VersionRequested* и *ImplementationDetails* используются командой WSAStartup для осуществления инициализации сети.

Отметим, что в программе сервера имеются две гнездовые переменные. Одна из них используется для хранения значения оконечного пункта, используемого для прослушивания новых клиентов. Другая хранит значение оконечного пункта текущего клиента.

Отметим также, что существуют две переменные SOCKADDR_IN. Одна используется для установления адреса оконечного пункта сервера. Другая заполняется адресом текущего клиента, когда он принимается сервером.

Три переменных установки дескриптора файлов (*fd_set*) используются, как если бы они были массивами гнезд. Они действительно представляют собой набор гнезд. Вы можете вставлять в *fd_set* новые гнезда и убирать оттуда имеющиеся. Эти переменные передаются в функцию *select()*. Дескриптор *fd_set AllSockets* используется для прослеживания всех активных гнезд, которые должны контролироваться системой.

```
FD_ZERO(&AllSockets);
FD_ZERO(&ReadSockets);
FD_ZERO(&ExceptSockets);

if ((argc<=1)||(argc>5))
  {printf("Usage: %s port-number [wait-flag] [log-flag] [Process-flag]\n",argv[0]);
  printf(" if port = 0 then BIND will choose port number\n");
  printf(" if wait-flag is 1 [default is 0], then user must hit return to recv\n");
  printf(" if log-flag is 1 [default is 0], then detail log messages appear\n");
   printf(" if process-flag is 1 [default is 1], then all exceptions are processed first\n");
     return(0);
    }

port   = atoi(argv[1]);
if (2<argc) WaitFlag = atoi(argv[2]);
else       WaitFlag = 0;
if (3<argc) LogFlag  = atoi(argv[3]);
else       LogFlag  = 0;
if (4<argc) ProcessAllExceptions   = atoi(argv[4]);
else       ProcessAllExceptions   = 1;
```

Инициализация программы

Макрокоманда FD_ZERO используется для инициализации fd_set. Все три переменные fd_set обнуляются, чтобы они не имели гнезд внутри набора.

Сервер имеет во многом ту же логику, что и клиент в фазе инициализации программы. Если имеется неправильное количество аргументов, сервер выдает сообщение об использовании и выходит из работы.

Если представлено правильное количество аргументов, аргументы считываются в соответствующие переменные. Если опциональные аргументы не даются, то используются значения по умолчанию.

```
printf("Server Echo Program 1.0\n\n");

VersionRequested = MAKEWORD(1,1);
rv = WSAStartup (VersionRequested,   &ImplementationDetails);
if (0 != rv ) ErrorStop ();
```

Инициализация сети

Это действительный код программы для инициализации сети в библиотеке WINSOCK. Если эта программа переносится на другую платформу, которая не имеет WINSOCK, тогда этот раздел не нужен.

```
ServerSocket = socket (PF_INET,SOCK_STREAM,0);
if (INVALID_SOCKET == ServerSocket) ErrorStop ();
ServerSocketAddress.sin_family          = AF_INET;                  // InterNet
ServerSocketAddress.sin_addr.S_un.S_addr  = INADDR_ANY;             // Any Address
ServerSocketAddress.sin_port            = port;
rv = bind (ServerSocket,(LPSOCKADDR)&ServerSocketAddress,sizeof(ServerSocketAddress));
if (0 != rv ) ErrorStop ();
rv = listen (ServerSocket,5);
if (0!= rv) ErrorStop ();

FD_SET(ServerSocket,&AllSockets);
```

Описание гнезда сервера

Этот раздел кода программы выполняет следующие функции:

- Создает оконечный пункт сервера.
- Соединяет оконечный пункт с конкретным портом.
- Адрес INADDR_ANY позволяет серверу привязываться к любому адресу в системе с множественной адресацией (a multihomed system). Система с множественной адресацией действует как шлюз

для соединения двух сетей; эта система имеет адрес в каждой сети. INADDR_ANY только осуществляет привязку к одному из адресов. Если вы не находитесь в системе с множественной адресацией, тогда осуществляется привязка к единственному адресу, который доступен, т.е. к адресу сервера.

- Функция *bind()* осуществляет привязку адреса, определенного в структуре *ServerAddress*, к конкретному гнезду.
- Функция *listen()* осуществляет установку очереди клиентов. Когда клиент пытается осуществить подключение к серверу, он получает автоматическое подтверждение и клиент не блокируется в функции *connect*. Если сервер не осуществляет прослушивание, то клиент получает от *connect* сообщение о сетевой ошибке. Хотя функция *listen()* позволяет клиенту продолжить обработку, она не устанавливает соединения.
- Прослушивающее гнездо вкладывается в fd_set *AllSockets*. Это гнездо будет единственным, которое находится в этом наборе. Этот набор представляет собой список всех контролируемых гнезд. Гнездо прослушивания требует контроля для клиентов, пытающихся осуществить соединение. Если на этом гнезде имеется активность, тогда вам нужно принять клиента, который пытается осуществить соединение.

```
Server Stop Flag = 0;
 while (0==Server Stop Flag)
   {
      UserFd_copy(&ReadSockets,&AllSockets);
      UserFd_copy(&ExceptSockets,&AllSockets);

        if (1==WaitFlag)
         {
          printf("Press Return to select for data from %s\n",NotMe);
          gets(buffer);
         }
      Irv = select (0,&ReadSockets,NULL,&ExceptSockets,NULL);
      if (SOCKET_ERROR == Irv) ErrorStop ();
```

Контроль активности

Это раздел программы сервера, где вы начинаете осуществлять обработку каждого из клиентов. Основной цикл сервера вводится установкой флага выхода (exit flag) на 0: так что сервер не будет выходить из обработки. Для инициализации двух наборов fd_sets — *ReadSockets* и *ExceptSockets* — вызывается функция *UserFd_copy*, написанная пользователем. Целью этих двух наборов fd_sets является осуществление контроля над всеми имеющимися гнездами относительно активности

считывания и активности исключительных сообщений.Набор fd_set *AllSockets* используется для инициализации этих двух наборов fd_sets.

Если установлен флаг *WaitFlag*, то пользователь приглашается к нажатию <Return> до вызова *select()*. Функция *select* вызывается для осуществления контроля над гнездами на предмет их активности. Эта функция блокируется до тех пор, пока на контролируемом гнезде не будет соответствующей активности. Первый параметр включается для достижения совместимости, и на него можно не обращать внимания. Следующие три параметра являются адресами трех наборов fd_sets. Первый fd_set содержит гнезда, подлежащие контролю на предмет активности считывания. Отметим, что этот набор был инициализирован для того, чтобы он содержал все контролируемые гнезда, поэтому все гнезда будут контролироваться на активность считывания. Следующий fd_set содержит гнезда, подлежащие контролю на предмет возможности записи. Это означает, что если вы пытаетесь записать в гнездо, вы не будете блокироваться. Так как вы не заинтересованы в осуществлении записи в какое-либо гнездо, этот параметр определяется как пустой (NULL), что означает, что вы не будете осуществлять контроль на предмет возможности записи. Последний параметр fd_set представляет гнезда, контролируемые на предмет исключений. Исключение на гнезде означает, что на этом гнезде было получено внедиапазонное (Out-of-Band) сообщение.

Последний параметр для *select* представляет собой значение выдержки (time-out). Вы переходите в NULL для указания о том, что вы не желаете осуществлять никакую выдержку. Функция *select* будет ждать до тех пор, пока не будет иметься нормальных данных, доступных для считывания, или пока не будет доступных внедиапазонных данных. Когда функция *select* осуществляет возвращение, набор *ReadSockets* fd_set будет содержать только гнезда, которые имеют доступные для считывания данные, а *ExceptSocket* fd_set будет содержать только гнезда, которые имеют доступные внедиапазонные данные. Если клиент пытается осуществить соединение на гнезде прослушивания, это состояние рассматривается как наличие данных, доступных для считывания, и сервер должен выполнить *accept* (прием).

```
if (FD_ISSET(ServerSocket,&ReadSockets))
  {
    ClientSocket = accept(ServerSocket,(LPSOCKADDR) &ClientSocketAddress,NULL);
    if (INVALID_SOCKET == ClientSocket) ErrorStop ();
    printf("Client (%d) attached\n",ClientSocket);
    FD_SET(ClientSocket,&AllSockets);
    FD_CLR(ServerSocket,&ReadSockets);
  }
```

Обработка подключающихся клиентов

Когда клиент пытается соединиться с сервером, возвращается функция *select* и одно из гнезд в наборе *ReadSockets* fd_set будет являться гнездом прослушивания. Переменная *ServerSocket* была первоначально загружена с гнездовым номером для того гнезда, которое прослушивает клиентов. Макрокоманда FD_ISSET используется для проверки того, находится ли номер *ServerSocket* в настоящее время в наборе *ReadSockets* fd_set. Если это гнездо находится в *ReadSockets* fd_set, то выполняются следующие шаги:

1. Вызывается функция *accept* для приема нового гнезда для клиента.

2. Если возникает ошибка, тогда сервер вызывает *ErrorStop*.

3. На печать выводится сообщение для указания того, что новый клиент подключен к серверу.

4. Новое гнездо добавляется к списку всех гнезд для контроля момента, когда функция *select* будет вызываться снова. Функция FD_SET используется для ввода нового гнезда в *AllSockets* fd_set.

5. *ServerSocket* убирается из списка гнезд, доступных для считывания, т.е. *ReadSockets*. Может случиться так, что клиент попытается подключиться к серверу, в то время как другие клиенты посылают данные на сервер. В этом случае оба гнезда будут находиться в наборе *ReadSockets* fd_set. Если вы только что обработали подключившегося клиента, то он устраняется из списка fd_set, и теперь вы можете обрабатывать других клиентов.

```
int UserExceptEach(fd_set *in,fd_set *all)
 {
  unsigned int i,j=0;
  int rv;
  ReadFlag = MSG_OOB;
  for (i=0;i<in->fd_count;i++)
   {
     printf("**** (%d) Out of Band Data\n",in->fd_array[i]);
     rv = UserReadClient(in->fd_array[i]);
     if (0==rv)
        { printf("(%d) Socket Error - Socket being closed\n",in- >fd_array[i]);
          FD_CLR(in->fd_array[i],all);
          closesocket(in->fd_array[i]);
        }
     j++;
     printf("**** End of this Out of Band packet\n");
   }
```

```
 ReadFlag = 0;
 return(j);
}
```

Обработка исключительных данных

Функция *UserExceptEach* осуществляет цикл над каждым гнездом в наборе fd_set, который передан в функцию. Она вызывает функцию *UserReadClient*, для считывания доступных данных на гнезде. До начала циклического процесса глобальная переменная *ReadFlag* устанавливается в MSG_OOB, так что функция *UserReadClient* будет считывать данные OOB, а не нормальные данные.

Эта функция использует тот факт, что набор fd_set реализован как массив в библиотеке WINSOCK. Если вы переносите этот сервер на другую платформу, вы можете заново проверить эту фунцию, если fd_set реализован по-другому.

После того как все гнезда обработаны, глобальная переменная *ReadFlag* сбрасывается, так что новые вызовы *UserReadClient* будут считывать нормальные данные.

Если при считывании данных возникает ошибка, то гнездо закрывается и изымается из списка гнезд, контролируемых макрокомандой FD_CLR.

```
int UserReadEach(fd_set *in,fd_set *all)
 {
  unsigned int i;
  int rv;
  for (i=0;i<in->fd_count;i++)
   {
    if (1==LogFlag) printf("Data Ready on Socket (%d)\n",in- >fd_array[i]);
    rv = UserReadClient(in->fd_array[i]);
    if (0==rv)
      { printf("(%d) Socket Error - Socket being closed\n",in- >fd_array[i]);
        FD_CLR(in->fd_array[i],all);
        closesocket(in->fd_array[i]);
      }
    if (-1==rv) return(1); //  Stop the server
   }
 return(0); // continue the server
}
```

Обработка нормальных данных

Функция *UserReadEach* осуществляет цикл над каждым из гнезд в наборе fd_set, который передан в функцию, и вызывает функцию *UserReadClient* для считывания доступных данных на гнезде.

Функция также использует тот факт, что набор fd_set реализован как массив в библиотеке WINSOCK. Если вы переносите этот сервер на другую платформу, вам, возможно, потребуется заново проверить эту функцию, если набор fd_set реализован по-другому.

Если при считывании данных возникает ошибка, то гнездо закрывается и изымается из списка гнезд, контролируемых макрокомандой FD_CLR.

Если значение возврата из *UserReadClient* равно -1, то от клиента поступает команда *quit*, и сервер должен остановиться. Функция UserReadEach возвратит значение 1 для указания того, что сервер должен остановиться. В противном случае после обработки всех клиентов она должна возвратить значение 0 для указания серверу продолжать работу.

```
int UserReadClient(SOCKET ClientSocket)
  {
        char        buffer[100];
        int         rv;

  rv = recv (ClientSocket,buffer,80,ReadFlag);
   if  (SOCKET_ERROR == rv)
     switch(WSAGetLastError())
       {  case(WSAECONNRESET):
          printf("(%d) Connection Reset\n",ClientSocket);
          return(0);  // signals error
         default: Error ();
       }
  if (0==rv)
      {printf("Zero Length Data interpreted as a network disconnect\n");
       printf("(%d)Connection Reset\n");
       return(0);  // signals  error
      }
   printf("socket:%03d>[%03d]",ClientSocket,rv);
  output(rv,buffer);   // output is used because buffer may contain NUL characters
```

Получение данных от клиента

Эта функция вызывает функцию гнезда *recv()* для приема до 80 знаков, которые ожидают во входном буфере гнезда. Глобальная переменная *ReadFlag* используется для определения того, какой осуществляется тип считывания нормальных данных или внедиапазонных (OOB) данных. После того как данные считаны, вы осуществляете проверку на наличие двух типов ошибок, которые, возможно, имели место. Если имела место ошибка, вы возвращаете нулевое значение на вызывающее устройство, для того чтобы оно изъяло гнездо из набора fd_set. Если же ошибок не было, то тогда вы выводите полученные данные на печать с помощью функции *output*.

Функция *output* позволяет выводить на дисплей в кавычках знаки, которые обычно на печать выведены быть не могут.

```
if (0==strcmp("exit",buffer))
  { printf("(%d)Client sent Exit command\n",ClientSocket);
    return(0); // signals error
  }

if (0==strcmp("quit",buffer))
  { printf("(%d)Client sent quit command, Server exiting...\n",ClientSocket);
    return(-1); // signals to exit the server
  }
```

Проверка команд Exit (выход)

После того как данные получены, они проверяются на предмет того, нет ли каких-либо встроенных команд, указывающих на выходящего из работы клиента. Команда *exit* указывает, что клиент вышел из работы. Сервер должен закрыть гнездо и изъять его из списка всех гнезд.

Команда *quit* дает серверу указание выйти из работы.

```
if (0==strcmp("slog",buffer))
      { LogFlag = (1==LogFlag) ? 0: 1;
        printf("Client sent slog command. LogFlag now %d\n",LogFlag);
        return(1); // signals normal
      }
if (0==strcmp("swait",buffer))
      { WaitFlag = (1==WaitFlag) ? 0: 1;
        printf("Client sent swait command. WaitFlag now %d\n",WaitFlag);
        return(1); // signals normal
      }
if (0==strcmp("sall",buffer))
      { ProcessAllExceptions = (1==ProcessAllExceptions) ? 0: 1;
        printf("Client sent sall command. ProcessAllExceptions now %d\n",ProcessAllExceptions);
       return(1);    // signals normal
      }
     return(1);        // signals normal
  }
```

Проверка команд переброса

Сервер осуществляет проверку трех команд переброса. Каждая из этих команд перебрасывает одну из глобальных переменных каждый раз, когда она принимается:

- *slog* — перебрасывает переменную *LogFlag*;
- *swait* — перебрасывает переменную *WaitFlag*;
- *sall* — перебрасывает переменную *ProcessAllExceptions*.

```
void UserFd_copy(fd_set *out, fd_set *in)
  {
  unsigned int i;

  for (i=0;i<in->fd_count;i++) out->fd_array[i]=in->fd_array[i];
  out->fd_count = in->fd_count;
  }
```

Копирование fd_set

Эта функция помощника (helper) предназначена для копирования одного набора fd_set в другой. Она зависит от знания внутренней реализации fd_set. Для каждого члена элемента массива fd_set вы копируете *входящие* (*in*) данные в *исходящие* (*out*) данные. Затем вы копируете значение счета из *входящей* (*in*) структуры в *исходящую* (*out*) структуру.

USERLIB.C

Это тот же файл, который описан в примере ECHO, со включением следующих операторов:

```
#undef ioctlsocket
#undef select

int UserLog_ioctlsocket(SOCKET ServerSocket,long cmd, unsigned long *argptr)
  {
    int err;
      if (1==LogFlag) printf("Calling ioctlsocket(SOCKET,%lx,- >[%lu]) to set non blocking
mode",cmd,*argptr);

    err = ioctlsocket(ServerSocket,cmd,argptr);
    if (SOCKET_ERROR==err) UserErrorMessage("ioctlsocket failed\n");
    else if (1==LogFlag) printf("            ioctlsocket succeeded\n\n");
    return(err);
  }

long UserLog_select(int ignore, fd_set *readfs, fd_set *writefs, fd_set *errorfs, const struct
timeval FAR * time-out)
  { long rv;
    if (1==LogFlag)
     {
     printf("Calling Select to see if any activity is available\n");
      printf(" This call will BLOCK until at least one of the sockets is ready\n");
     }
    rv = select(ignore,readfs,writefs,errorfs,time-out);
    if (SOCKET_ERROR == rv) UserErrorMessage("Select Failed\n");
    else if (1==LogFlag) printf("Select Returned (%ld)\n",rv);
    return(rv);
  }
```

СОЗДАНИЕ ГОТОВЫХ К ВЫПОЛНЕНИЮ ПРОГРАММ

Компилирование под NT SDK

Для компилирования и компоновки исходных файлов с использованием комплекта средств разработчика программного обеспечения Windows NT используйте следующий формирующий файл:

```
!include <ntwin32.mak>
cflags=$(cflags) -DDEVL=1 -D_MT=1 -Gz
all: server.exe client.exe
server.exe: server.obj userlib.obj $(link) $(linkdebug) $(conflags) -out:$*.exe $** $(conlibsmt)
wsock32.lib
client.exe: client.obj userlib.obj $(link) $(linkdebug) $(conflags) -out:$*.exe $**  $(conlibsmt)
wsock32.lib
client.obj: client.c userlib.h $(cc) $(cflags) $(cvarsmt) $(cdebug) client.c
server.obj: server.c userlib.h $(cc) $(cflags) $(cvarsmt) $(cdebug) server.c
userlib.obj: userlib.c userlib.h $(cc) $(cflags) $(cvarsmt) $(cdebug) userlib.c
```

Этот формирующий файл описывает, как создать следующие три файла .OBJ:

- CLIENT.OBJ — зависит от CLIENT.C и USERLIB.H,
- SERVER.OBJ — зависит от SERVER.C и USERLIB.H,
- USERLIB.OBJ — зависит от USERLIB.C и USERLIB.H.

Два файла .EXE создаются следующим образом:

- SERVER.EXE — зависит от SERVER.OBJ и USERLIB.OBJ,
- CLIENT.EXE — зависит от CLIENT.OBJ и USERLIB.OBJ.

Осуществляйте компоновку как стандартной консольной программы *$(conflags)* и компонуйте со следующими библиотеками:

- многонитевыми (multithreaded) консольными библиотеками *$(conlibsmt)*,
- библиотекой гнезд Windows WINSOCK32.LIB.

ИСПОЛЬЗОВАНИЕ VISUAL C++

Создание файла нового проекта и построение готовых к выполнению программ (executables)

Для создания нового проекта в Visual C++ используйте шаги, описанные в примере ECHO. Нижеприведенный список определяет эти шаги:

- Используйте меню New Project для создания нового проекта.
- Определите проект как Console Application или QuickWin, если вы используете Version 3.1 Windows.

- Выберите файлы Server.c и Userlib.c для добавления их к проекту.
- Установите опции Project с помощью меню Options Project.
- Выберите диалоговое окно Linker...
- Выберите опции Input.
- Добавьте WSOCK32.LIB — или WINSOCK.LIB для Windows Version 3.1 — к модулям библиотеки (Library modules).
- Постройте исполнительный файл, щелкнув мышью по клавише Build.
- Установите аргументы командной строки с помощью элемента Options Debug: 100 1 1 1.
- Осуществите те же шаги, которые описаны выше, и добавьте Client.c и Userlib.c к другому проекту и постройте исполняемую программу.
- Установите аргументы командной строки на 127.0.0.1. 100 1 1.

Эталонная программа

В следующем примере стартует сервер и пять клиентов, причем старты осуществляются в различные моменты времени. Рассматриваются некоторые из интересных побочных эффектов согласования во времени и сообщения упорядочения.

РЕГИСТРАЦИЯ СЕРВЕРА (С ПОЯСНЕНИЯМИ)

```
Server Echo Program 1.0
   Calling WSAStartup(Version,Details) to locate a Windows DLL
   WSAStartup has returned ImplementationDetails:
     wVersion:   (257)
     wHighVersion: (257)
     szDescription:'Microsoft Windows Sockets Version 1.1.'
     szSytemStatus:'Running.'
     iMaxSockets: (32767)
     iMaxUdpDg:  (65467)
     lpVendorInfo (000202AD)

   Calling socket(AddressFormat,Type,Protocol) to Create a Server Socket
   Server Socket (56) has been created

   Calling bind(Socket,AddressPointer,length) to Bind the socket
   Bind succeeded

   Calling listen(Socket,MaxConnections)
   Successfully changed to a listen socket

Calling Select to see if any activity is available
  This call will BLOCK until at least one of the sockets is ready
```

```
( Server Echo Program 1.0
  Вызов WSAStartup(версия,детали) для определения места Windows DLL
  WSAStartup возвратила детали реализации:
        wVersion: (257)
        wHighVersion: (257)
        szDescription: 'Microsoft Windows Sockets Version 1.1'
        szSystemStatus: 'Running' iMaxSockets: (32767)
        iMaxUdpDg: (65467)
        lpVendorInfo: (000202AD)

  Вызов гнезда(AddressFormat,Type,Protocol) для создания гнезда сервера
  Гнездо сервера (56) создано

  Вызов bind(Socket,AddressPointer,length) для привязывания гнезда
  Bind прошла успешно

  Вызов Listen(Socket,MaxConnections)
  Осуществлено успешное внесение изменений в гнездо прослушивания

  Вызов Select для поиска активности
    Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово  )
```

Клиент 1 стартовал

```
 Select Returned (1)
     Calling accept(ListenSocket,AddressPointer,length)
     This function will BLOCK until a client is available
     accept succeeded: A client is available

 Client (36) attached
 Calling Select to see if any activity is available
   This call will BLOCK until at least one of the sockets is ready

( Select Returned (1)
        Вызов accept(ListenSocket,AddressPointer,length)
        Эта функция будет БЛОКИРОВАТЬСЯ, пока клиент не станет доступен
        accept прошла успешно: клиент доступен

Клиент(36) подключен
Вызов Select для поиска активности
    Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово  )
```

Данные, введенные у клиента 1: {1}C1:M1

```
 Select Returned (1)
 Data Ready on Socket (36)
  socket:036>[009]{1}C1:M1\000
 Calling Select to see if any activity is available
   This call will BLOCK until at least one of the sockets is ready
```

(Select Returned (1)
Данные готовы на Гнезде (36)
socket:036>[009]{1}C1:M1\000
Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Данные вводятся у клиента 1:{2}C1:M2

Select Returned (1)
Data Ready on Socket (36)
socket:036>[009]{2}C1:M2\000
Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready

(Select Returned (1)
Данные готовы на Гнезде (36)
socket:036>[009]{2}C1:M2\000
Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Клиент 2 стартовал

Select Returned (1)
Calling accept(ListenSocket,AddressPointer,length)
This function will BLOCK until a client is available
accept succeeded: A client is available

Client (16) attached
Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready

(Select Returned (1)
Вызов accept(ListenSocket,AddressPointer,length)
Эта функция будет БЛОКИРОВАТЬСЯ, пока клиент не станет доступен
accept прошла успешно: Клиент доступен

Клиент (16) подключен
Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Данные, введенные у Клиента 2:{3}C2:M1

Select Returned (1)
Data Ready on Socket (16)
socket:016>[009]{3}C2:M1\000
Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready

(Select Returned (1)

Данные готовы на Гнезде (16)
socket:016>[009]{3}C2:M1\000
Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Данные, введенные у Клиента 2:{4}C2:M2

Select Returned (1)
Data Ready on Socket (16)
socket:016>[009]{4}C2:M2\000
Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready

(Select Returned (1)
Данные готовы на Гнезде (16)
socket:016>[009]{4}C2:M2\000
Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Клиент 3 стартовал

Select Returned (1)
Calling accept(ListenSocket,AddressPointer,length)
This function will BLOCK until a client is available
accept succeeded: A client is available

Client (284) attached
Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready

(Select Returned (1)
Вызов accept(ListenSocket,AddressPointer,length)
Эта функция будет БЛОКИРОВАТЬСЯ, пока клиент не станет доступен
accept прошла успешно: Клиент доступен

Клиент (284) подключен
Вызов Select для проверки активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Данные, введенные у Клиента 3:{5}C3:M1

Select Returned (1)
Data Ready on Socket (284)
socket:284>[009]{5}C3:M1\000
Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready

(Select Returned (1)
Данные готовы на Гнезде (284)

socket:284>[009]{5}C3:M1\000
Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Данные, введенные у Клиента 3:{6}C3:M2

Select Returned (1)
Data Ready on Socket (284)
socket:284>[009]{6}C3:M2\000
Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready

(Select Returned (1)
Данные готовы на Гнезде (284)
socket:284>[009]{6}C3:M2\000
Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Данные, введенные у Клиента 3: slog

Select Returned (1)
Data Ready on Socket (284)
socket:284>[005]slog\000
Client sent slog command. LogFlag now 0

(Select Returned (1)
Данные готовы на Гнезде (284)
socket:284>[005]slog\000
Клиент послал команду slog. Теперь LogFlag равен 0)
[Регистрация отключена]
Данные, введенные у Клиента 3:{7}C3:M3

socket:284>[009](7)C3:M3\000

Данные, введенные у Клиента 2: swait

socket:016>[006]swait\000
Client sent swait command. WaitFlag now 1
Press Return to select for data from Client

(socket:016>[006]swait\000
Клиент послал команду swait. Теперь WaitFlag равен 1
Нажмите Return для выбора данных у Клиента)

Данные, введенные у Клиента 2:{8}C2:M3
Данные, введенные у Клиента 1:{9}C1:M3
Данные, введенные у Клиента 3: slog
Данные, введенные у Клиента 1:{10}C2:M4

Данные, введенные у Клиента 2:{11}C2:M4
<Return на сервере осуществляет вызов Select>
Обработка первого гнезда (Клиент 1)
Обратите внимание на то, что вы получаете все данные, ожидающие на гнезде, а не только первое сообщение.

socket:036>[019](9)C1:M3\000{10}C1:M4\000
Обработка второго гнезда (Клиент 2)

socket:016>[020]{8}C2:M3\000{11r}C2:M4\000
Обработка третьего гнезда (Клиент 3)

socket:284>[005]slog\000
Client sent slog command. LogFlag now 1
(Клиент послал команду slog. Теперь LogFlag = 1)
[Регистрация задействована снова]

Press Return to select for data from Client
(Нажмите Return для выбора данных от Клиента)

Данные, введенные у Клиента 1:coob
[Клиент 1 теперь начнет отсылку данных OOB]
Данные OOB, введенные у Клиента 1:{12}C1:U1
Данные, введенные у Клиента 2:{13}C2:M5
Данные, введенные у Клиента 3:{14}C3:M4
Данные, введенные у Клиента 3: coob
[Клиент 3 теперь начнет отсылку данных OOB]
Данные OOB, введенные у Клиента 3:{15u}C3:U1
Данные OOB, введенные у Клиента 1:{16u}C1:U2
Данные, введенные у Клиента 2:{17}C2:M6
Данные OOB, введенные у Клиента 3:{18ur}C3:U2
<Return на сервере осуществляет вызов Select>

Calling Select to see if any activity is available
(Вызов Select чтобы посмотреть, имеется ли какая-либо активность)

This call will BLOCK until at least one of the sockets is ready
(Этот вызов будет БЛОКИРОВАН до тех пор, пока хотя бы одно из гнезд не будет готово)
Select Returned (5)
2 гнезда имеют данные OOB, а три гнезда имеют нормальные данные
Обработка данных OOB для первого исключительного гнезда (Клиент 1)

**** (36) Out of Band Data
socket:036>[022]{12u}C1:U1\000{16u}C1:U2\000
**** End of this Out of Band packet

```
( **** (36) Внедиапазонные данные
 socket:036>[022]{12u}C1:U2\000
 **** Конец этих внедиапазонных данных  )
```

Обработка данных OOB для второго исключительного гнезда (Клиент 3)

```
 **** (284) Out of Band Data
 socket:284>[023]{15u}C3:U1\000{18ur}C3:U2\000
 **** End of this Out of Band packet

( **** (284) Внедиапазонные данные
 socket:284>[023]{15u}C3:U1\000{18ur}C3:U2\000
 **** Конец этого внедиапазонного пакета  )
```

Так как вы имеете исключительные данные и установлен флаг *ProcessAllExceptions*, вы не обрабатываете нормальные данные. Вы возвращаетесь для поиска новых исключительных данных.

```
Press Return to select for data from Client
(Нажмите Return для поиска данных от Клиента)
```

Данные OOB, введенные у Клиента 1: coob
[Теперь клиент 1 начнет посылать нормальные данные]
Данные, введенные у Клиента 2:{19}C2:M7
Данные OOB, введенные у Клиента 3: coob
[Теперь Клиент 3 начнет посылать нормальные данные]
Данные, введенные у Клиента 1:{20}C1:M5
Данные, введенные у Клиента 2:{21}C2:M8
Данные, введенные у Клиента 3:{22r}C3:M5
<Return на сервере осуществляет вызов Select>

```
Calling Select to see if any activity is available
(Вызов Select для поиска активности)

  This call will BLOCK until at least one of the sockets is ready
  (Этот вызов будет БЛОКИРОВАТЬСЯ до тех пор, пока хотя бы одно из гнезд не будет готово)
 Select Returned (5)
```

2 гнезда имеют данные OOB, а 3 гнезда имеют нормальные данные.
Обработка данных OOB из первого исключительного пакета (Клиент 1)

```
 **** (36) Out of Band Data
 socket:036>[005]coob\000
 **** End of this Out of Band packet

( **** (36) Внедиапазонные данные
 socket:284>[005]coob\000
 **** Конец этого внедиапазонного пакета )
```

Обработка данных OOB для второго исключительного пакета (Клиент 3)

```
**** (284) Out of Band Data
socket:284>[005]coob\000
**** End of this Out of Band packet

( **** (284) Внедиапазонные данные
socket:284>[005]coob\000
**** Конец этого внедиапазонного пакета )
```

Так как вы имеете исключительные данные и флаг *ProcessAllExceptions* установлен, вы не обрабатываете нормальные данные. Вы возвращаетесь для поиска новых исключительных данных.

```
Press Return to select data from Client
(Нажмите Return для выбора данных от Клиента)
Данные, введенные у Клиента 1:{23}C1:M6
Данные, введенные у Клиента 3:{24г}C3:M6
<Return на сервере осуществляет вызов Select>

Calling Select to see if any activity is available
(Вызов Select для поиска активности)

 This call will BLOCK until at least one of the sockets is ready
 (Этот вызов будет БЛОКИРОВАТЬСЯ до тех пор, пока хотя бы одно из гнезд не будет готово)
Select Returned (3)
0 гнезд OOB и 3 нормальных гнезда
Обработка Гнезда 1 (Клиент 1) — здесь coob — это первый coob, который включил
режим OOB

Data Ready on Socket (36)
socket:036>[025]coob\000{20}C1:M5\000{23}C1:M6\000

( Данные готовы на Гнезде (36)
socket:036>[025]coob\000{20}C1:M5\000[23]C1:M6\000 )
Обработка Гнезда 2 (Клиент 2)

Data Ready on Socket (16)
socket:016>[040]{13}C2:M5\000{17}C2:M6\000{19}C2:M7\000{21}C2:M8\000R

( Данные готовы на Гнезде (16)
socket:016>[040]{13}C2:M5\000{17}C2:M6\000{19}C2:M7\000{21}C2:M8\000R )
Обработка Гнезда 3 (Клиент 3) — coob находится в середине первого coob, кото-
рый включил режим OOB.

Data Ready on Socket (284)
socket:284>[037]{14}C3:M4\000coob\000{22г}C3:M5\000{24г}C3:M6\000
Press Return to select for data from Client

( Данные готовы на Гнезде (284)
```

socket:284>[037]{14}C3:M4\000coob\000{22r}C3:M5\000{24r}C3:M6\000
Нажмите Return для выбора данных от Клиента)
Данные, введенные у Клиента 1:bye
[Клиент 1 выходит из обработки — Сервер не оповещается]
Данные, введенные у Клиента 2:{25}C2:M9
Клиент 4 стартует
Данные, введенные у Клиента 3:{26}C3:M7
Данные, введенные у Клиента 4:swait
Данные, введенные у Клиента 3:{27r}C3:M8
<Return на Сервере осуществляет вызов Select>

Calling Select to see if any activity is available
(Вызов Select для поиска активности)
Набор read fd_set содержит только три гнезда — Клиент 1, Клиент 2, Клиент 3

This call will BLOCK until at least one of the sockets is ready
(Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)
Select Returned (4)
4 читаемых клиента, 1 требует приема, 3 требуют считывания

Calling accept(ListenSocket,AddressPointer,length)
This function will BLOCK until a client is available
accept succeeded: A client is available

Client (264) attached

(Вызов accept (ListenSocket,AddressPointer,length)
Эта функция будет БЛОКИРОВАТЬСЯ до тех пор, пока клиент не станет доступен
Команда accept прошла: клиент доступен

Клиент (264) подключился)
Обработка Гнезда 1 (Клиент 1)

Data Ready on Socket (36)
socket:036>[004]bye\000

(Данные готовы на Гнезде (36)
socket:036>[004]bye\000)
Обработка гнезда 2 (Клиент 2)

Data Ready on Socket (16)
socket:016>[010]{25}C2:M9\000

(Данные готовы на Гнезде (16)
socket:016>[010]{25}C2:M9\000)
Обработка Гнезда 3 (Клиент 3)

Data Ready on Socket (284)
socket:284>[021]{26}C3:M7\000{27r}C3:M8\000

```
( Данные готовы на Гнезде (284)
socket:284>[021]{26}C3:M7\000{27r}C3:M8\000   )
```

Так как список гнезд в обрабатываемом наборе fd_set был создан ДО функции *accept*, любые ожидающие данные у Клиента 4 не обрабатываются. Гнездо для Клиента 4 было введено в набор *AllSockets* fd_set и будет контролироваться, когда функция *select* будет вызвана в следующий раз.

```
Press Return to select the data from Client
(Нажмите Return для выбора данных от Клиента)
```

Данные, введенные у Клиента 2: exit
Данные, введенные у Клиента 3:{28}C3:M9
Данные, введенные у Клиента 4: swait
<Return на сервере осуществляет вызов Select>

```
Calling Select to see if any activity is available
  This call will BLOCK until at least one of the sockets is ready
Select Returned (4)
Data Ready on Socket (36)
(36) Connection Reset
(36) Socket Error - Socket being closed
Data Ready on Socket (16)
socket:016>[005]exit\000
(16) Client sent Exit command
(16) Socket Error - Socket being closed
Data Ready on Socket (284)
socket:284>[011]{28}C3:M9\000
Data Ready on Socket (264)
socket:264>[006]swait\000
Client sent swait command. WaitFlag now 0
Calling Select to see if any activity is available
  This call will BLOCK until at least one of the sockets is ready

( Вызов Select для поиска активности
  Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово
Select Returned (4)
Данные готовы на Гнезде (36)
(36) Сброс соединения
(36) Ошибка гнезда — Гнездо закрывается
Данные готовы на Гнезде (16)
socket:016>[005]exit\000
(16) Клиент послал команду Exit
(16) Ошибка гнезда — Гнездо закрывается
Данные готовы на Гнезде (284)
socket:284>[011]{28}C3:M9\000
Данные готовы на Гнезде (264)
```

socket:264>[006]swait\000
Клиент послал команду swait.Теперь WaitFlag — 0.
Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Данные, введенные у Клиента 3: exit

Select Returned (1)
Data Ready on Socket (284)
socket:284>[005]exit\000
(284)Client sent Exit command
(284) Socket Error - Socket being closed
Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready

(Select Returned (1)
Данные готовы на Гнезде (284)
socket:284>[005]exit\000
(284)Клиент послал команду Exit
(284) Ошибка гнезда — Гнездо закрывается
Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово)

Данные, введенные у Клиента 4: swait

Select Returned (1)
Data Ready on Socket (264)
socket:264>[006]swait\000
Client sent swait command. WaitFlag now 1
Press Return to select for data from Client

(Select Returned(1)
Данные готовы на Гнезде (264)
socket:264>[006]swait\000
Клиент послал команду swait. Теперь WaitFlag — 1
Нажмите Return для выбора данных от Клиента)

Клиент 5 стартовал

Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready
Select Returned (1)
Calling accept(ListenSocket,AddressPointer,length)
This function will BLOCK until a client is available
accept succeeded: A client is available
Client (244) attached
Press Return to select for data from Client

(Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово
Select Returned (1)
Вызов accept(ListenSocket,AddressPointer,length)

Эта функция будет БЛОКИРОВАТЬСЯ, пока хотя одно из гнезд не будет готово
accept прошла успешно: Клиент доступен для обработки
Клиент (244) подключен
Нажмите Return для выбора данных от Клиента)
Данные, введенные у Клиента 4:quit
Данные, введенные у Клиента 5:{28r}C5:M1
<Return на сервере осуществляет вызов Select>

Calling Select to see if any activity is available
This call will BLOCK until at least one of the sockets is ready
Select Returned (2)
Data Ready on Socket (264)
socket:264>[005]quit\000
(264)Client sent quit command, Server exiting...
Exiting the Server ....

(Вызов Select для поиска активности
Этот вызов будет БЛОКИРОВАТЬСЯ, пока хотя бы одно из гнезд не будет готово
Select Returned (2)
Данные готовы на Гнезде (264)
socket:264>[005]quit\000
(264) Клиент послал команду quit, Сервер выходит из обработки...
Сервер выходит из обработки...)
Данные для Клиента 5 не обрабатываются.

РЕГИСТРАЦИЯ КЛИЕНТА 1

Client Echo Program 1.0

Calling WSAStartup(Version,Details) to locate a Windows DLL
WSAStartup has returned ImplementationDetails:
wVersion: (257)
wHighVersion: (257)
szDescription:'Microsoft Windows Sockets Version 1.1.'
szSytemStatus:'Running.'
iMaxSockets: (32767)
iMaxUdpDg: (65467)
lpVendorInfo (00000010)

Calling socket(AddressFormat,Type,Protocol) to Create a Client Socket
Client Socket (56) has been created

Calling connect(ServerSocket,AddressPointer,length)
Connect succeeded: A Server is available

Enter data to transmit to the server
Client>**{1}C1:M1**
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

```
   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   9 characters sent to the Server
   Data: {1}C1:M1
Client>{2}C1:M2
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   9 characters sent to the Server
   Data: {2}C1:M2
Client>{9}C1:M3
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   9 characters sent to the Server
   Data: {9}C1:M3
Client>{10}C1:M4
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   10 characters sent to the Server
   Data: {10}C1:M4
Client>coob
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   5 characters sent to the Server
   Data: coob
Client noticed coob command. SendFlag now 1
Client>{12u}C1:U1
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   11 characters sent to the Server
   Data: {12u}C1:U1
Client>{16u}C1:U2
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   11 characters sent to the Server
   Data: {16u}C1:U2
Client> coob
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded
```

```
    Calling send(Socket,buffer,SizeOfBuffer,Flags)
    5 characters sent to the Server
    Data: coob
Client noticed coob command. SendFlag now 0
Client>{20}C1:M5
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

    Calling send(Socket,buffer,SizeOfBuffer,Flags)
    10 characters sent to the Server
    Data: {20}C1:M5
Client>{23}C1:M6
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

    Calling send(Socket,buffer,SizeOfBuffer,Flags)
    10 characters sent to the Server
    Data: {23}C1:M6
Client> bye
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

    Calling send(Socket,buffer,SizeOfBuffer,Flags)
    4 characters sent to the Server
    Data: bye
Client closing socket
press Return to Exit Client

( Вызов гнезда(AddressFormat,Type,Protocol) для создания гнезда клиента
Гнездо клиента (56) создано
Вызов connect(ServerSocket,AddressPointer,length)
Соединение прошло: Сервер доступен

Введите данные для передачи серверу
Client>{1]}C1:M1
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

    Вызов send(Socket,buffer,SizeOfBuffer,Flags)
    9 знаков посланы на сервер
    Данные: {1}C1:M1
Client>{2}C1:M2
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

    Вызов send(Socket,buffer,SizeOfBuffer,Flags)
    9 знаков посланы на сервер
    Данные: {1}C1:M2
Client>{9}C1:M3
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла
```

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
9 знаков посланы на сервер
Данные: {9}C1:M3
Client>{10}C1:M4
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
10 знаков посланы на сервер
Данные: {10}C1:M4
Client>coob
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
5 знаков посланы на сервер
Данные: coob
Клиент обнаружил команду coob. Теперь SendFlag = 1
Client>{12u}C1:U1
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
11 знаков посланы на сервер
Данные: {12u}C1:U1
Client>{16u}C1:U2
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
11 знаков посланы на сервер
Данные: {16u}C1:U2
Client>coob
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
5 знаков посланы на сервер
Данные: coob
Клиент обнаружил команду coob. Теперь SendFlag = 0
Client>{20}C1:M5
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
10 знаков посланы на сервер
Данные: {20}C1:M5
Client>{23}C1:M6
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

```
   Вызов send(Socket,buffer,SizeOfBuffer,Flags)
   10 знаков посланы на сервер
   Данные: {23}C1:M6
Client>bye
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

   Вызов send(Socket,buffer,SizeOfBuffer,Flags)
   4 знака посланы на сервер
   Данные: bye
Клиент закрывает гнездо
Нажмите Return для осуществления выхода клиента )
```

РЕГИСТРАЦИЯ КЛИЕНТА 2

```
Client Echo Program 1.0

   Calling WSAStartup(Version,Details) to locate a Windows DLL
   WSAStartup has returned ImplementationDetails:
     wVersion:   (257)
     wHighVersion: (257)
     szDescription:'Microsoft Windows Sockets Version 1.1.'
     szSytemStatus:'Running.'
     iMaxSockets: (32767)
     iMaxUdpDg:  (65467)
     lpVendorInfo (00000010)

   Calling socket(AddressFormat,Type,Protocol) to Create a Client Socket
   Client Socket (56) has been created

   Calling connect(ServerSocket,AddressPointer,length)
   Connect succeeded: A Server is available

Enter data to transmit to the server
Client> {3}C2:M1
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   9 characters sent to the Server
   Data: {3}C2:M1
Client> {4}C2:M2
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   9 characters sent to the Server
   Data: {4}C2:M2
```

```
Client> swait
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   6 characters sent to the Server
   Data: swait
Client> {8}C2:M3
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   9 characters sent to the Server
   Data: {8}C2:M3
Client> {11r}C2:M4
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   11 characters sent to the Server
   Data: {11r}C2:M4
Client> {13}C2:M5
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   10 characters sent to the Server
   Data: {13}C2:M5
Client> {17}C2:M6
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   10 characters sent to the Server
   Data: {17}C2:M6
Client> {19}C2:M7
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   10 characters sent to the Server
   Data: {19}C2:M7
Client> {21}C2:M8
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   10 characters sent to the Server
   Data: {21}C2:M8
Client> {25}C2:M9
```

```
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

    Calling send(Socket,buffer,SizeOfBuffer,Flags)
    10 characters sent to the Server
    Data: {25}C2:M9
Client> exit
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

    Calling send(Socket,buffer,SizeOfBuffer,Flags)
    5 characters sent to the Server
    Data: exit
Client sending exit command,Client exiting...

( Вызов гнезда(AddressFormat,Type,Protocol) для создания гнезда Клиента
Гнездо Клиента (56) создано

Вызов connect(ServerSocket,AddressPointer,length)
Соединение прошло: Сервер доступен

Ввод данных для передачи на сервер
Клиент> {3}C2:M1
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

    Вызов send(Socket,buffer,SizeOfBuffer,Flags)
    9 знаков посланы на сервер
    Данные: {3}C2:M1
Клиент> {4}C2:M2
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

    Вызов send(Socket,buffer,SizeOfBuffer,Flags)
    9 знаков посланы на сервер
    Данные: {3}C2:M2
Клиент> swait
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

    Вызов send(Socket,buffer,SizeOfBuffer,Flags)
    6 знаков посланы на сервер
    Данные: swait
Клиент> {8}C2:M3
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

    Вызов send(Socket,buffer,SizeOfBuffer,Flags)
    9 знаков посланы на сервер
    Данные: {8}C2:M3
```

Клиент> {11r}C2:M4
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
11 знаков посланы на сервер
Данные: {11r}C2:M4
Клиент> {13}C2:M5
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
10 знаков посланы на сервер
Данные: {13}C2:M5
Клиент> {17}C2:M6
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
10 знаков посланы на сервер
Данные: {17}C2:M6
Клиент> {19}C2:M7
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
10 знаков посланы на сервер
Данные: {19}C2:M7
Клиент> {21}C2:M8
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
10 знаков посланы на сервер
Данные: {21}C2:M8
Клиент> {25}C2:M9
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
10 знаков посланы на сервер
Данные: {25}C2:M9
Клиент> exit
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
5 знаков посланы на сервер
Данные: exit

Клиент посылает команду exit. Клиент выходит из обработки...)

РЕГИСТРАЦИЯ КЛИЕНТА 3

```
Client Echo Program 1.0

   Calling WSAStartup(Version,Details) to locate a Windows DLL
   WSAStartup has returned ImplementationDetails:
     wVersion:   (257)
     wHighVersion: (257)
     szDescription:'Microsoft Windows Sockets Version 1.1.'
     szSytemStatus:'Running.'
     iMaxSockets: (32767)
     iMaxUdpDg:  (65467)
     lpVendorInfo (00000010)

   Calling socket(AddressFormat,Type,Protocol) to Create a Client Socket
   Client Socket (56) has been created

   Calling connect(ServerSocket,AddressPointer,length)
   Connect succeeded: A Server is available

Enter data to transmit to the server
Client> {5}C3:M1
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   9 characters sent to the Server
   Data: {5}C3:M1
Client> {6}C3:M21
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   9 characters sent to the Server
   Data: {6}C3:M2
Client> slog
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   5 characters sent to the Server
   Data: slog
Client> {7}C3:M3
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   9 characters sent to the Server
   Data: {7}C3:M3
```

```
Client> slog
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   5 characters sent to the Server
   Data: slog
Client> {14}C3:M4
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   10 characters sent to the Server
   Data: {14}C3:M4
Client> coob
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   5 characters sent to the Server
   Data: coob
Client noticed coob command. SendFlag now 1
Client> {15u}C3:U1
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   11 characters sent to the Server
   Data: {15u}C3:U1
Client> {18u}C3:U2
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   12 characters sent to the Server
   Data: {18ur}C3:U2
Client> coob
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   5 characters sent to the Server
   Data: coob
Client noticed coob command. SendFlag now 0
Client> {22r}C3:M5
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   11 characters sent to the Server
   Data: {22r}C3:M5
```

```
Client> {24r}C3:M6
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   11 characters sent to the Server
   Data: {24r}C3:M6
Client> {26}C3:M7
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   10 characters sent to the Server
   Data: {26}C3:M7
Client> {27r}C3:M8
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   11 characters sent to the Server
   Data: {27r}C3:M8
Client> {28r}C3:M9
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   11 characters sent to the Server
   Data: {28r}C3:M9
Client> exit
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   5 characters sent to the Server
   Data: exit
Client sending exit command,Client exiting...

( Вызов гнезда (AddressFormat,Type,Protocol) для создания гнезда Клиента
Гнездо Клиента (56) создано

Вызов connect(ServerSocket,AddressPointer,length)
Соединение прошло: сервер доступен

Ввести данные для передачи на сервер
Клиент>{5}C3:M1
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

   Вызов send(Socket,buffer,SizeOfBuffer,Flags)
   9 знаков посланы на сервер
   Данные: {5}C3:M1
```

Клиент>{6}C3:M2
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
9 знаков посланы на сервер
Данные: {6}C3:M2
Клиент> slog
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
5 знаков посланы на сервер
Данные: slog
Клиент> {7}C3:M3
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
9 знаков посланы на сервер
Данные: {7}C3:M3
Клиент> slog
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
5 знаков посланы на сервер
Данные: slog
Клиент> {14}C3:M4
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
10 знаков посланы на сервер
Данные: {14}C3:M4
Клиент> coob
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
5 знаков посланы на сервер
Данные: coob
Клиент обнаружил команду coob. SendFlag теперь равен 1
Клиент> {15u}C3:U1
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
11 знаков посланы на сервер
Данные: {15u}C3:U1

Клиент> {18u}C3:U2
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
12 знаков посланы на сервер
Данные: {18u}C3:U2
Клиент> coob
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
5 знаков посланы на сервер
Данные: coob
Клиент обнаружил команду coob. SendFlag теперь равен 0
Клиент> {22r}C3:M5
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
11 знаков посланы на сервер
Данные: {22r}C3:M5
Клиент> {24r}C3:M6
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
11 знаков посланы на сервер
Данные: {24r}C3:M6
Клиент> {26}C3:M7
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
10 знаков посланы на сервер
Данные: {26}C3:M7
Клиент> {27r}C3:M8
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
11 знаков посланы на сервер
Данные: {27r}C3:M8
Клиент> {28r}C3:M9
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
11 знаков посланы на сервер
Данные: {28r}C3:M9

Клиент> exit
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
5 знаков посланы на сервер
Данные: exit
Клиент посылает команду exit. Клиент выходит из обработки...)

РЕГИСТРАЦИЯ КЛИЕНТА 4

Client Echo Program 1.0

Calling WSAStartup(Version,Details) to locate a Windows DLL
WSAStartup has returned ImplementationDetails:
wVersion: (257)
wHighVersion: (257)
szDescription:'Microsoft Windows Sockets Version 1.1.'
szSytemStatus:'Running.'
iMaxSockets: (32767)
iMaxUdpDg: (65467)
lpVendorInfo (00000010)

Calling socket(AddressFormat,Type,Protocol) to Create a Client Socket
Client Socket (56) has been created

Calling connect(ServerSocket,AddressPointer,length)
Connect succeeded: A Server is available

Enter data to transmit to the server
Client> swait
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

Calling send(Socket,buffer,SizeOfBuffer,Flags)
6 characters sent to the Server
Data: swait
Client> swait
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

Calling send(Socket,buffer,SizeOfBuffer,Flags)
6 characters sent to the Server
Data: swait
Client> quit
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

Calling send(Socket,buffer,SizeOfBuffer,Flags)
5 characters sent to the Server
Data: quit
Client sending quit command, Client exiting...

```
( Вызов WSAStartup(Version,Details) для определения места Windows DLL
WSAStartup возвратила детали реализации:
wVersion: (257)
wHighVersion: (257)
szDescription: 'Microsoft Windows Sockets Version 1.1'
szSystemStatus: 'Running'
iMaxSockets: (32767)
iMaxUdpDg: (65467)
lpVendorInfo (00000010)

Вызов гнезда(AddressFormat,Type,Protocol) для создания гнезда Клиента
Гнездо Клиента (56) создано

Вызов connect(ServerSocket,AddressPointer,length)
Connect прошла: Сервер доступен

Ввести данные для передачи на сервер
Клиент> swait
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

   Вызов send(Socket,buffer,SizeOfBuffer,Flags)
   6 знаков послано на сервер
   Данные: swait
Клиент> swait
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

   Вызов send(Socket,buffer,SizeOfBuffer,Flags)
   6 знаков послано на сервер
   Данные: swait
Клиент> quit
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

   Вызов send(Socket,buffer,SizeOfBuffer,Flags)
   5 знаков послано на сервер
   Данные: quit
Клиент посылает команду quit. Клиент выходит из обработки... )
```

РЕГИСТРАЦИЯ КЛИЕНТА 5

```
Client Echo Program 1.0

   Calling WSAStartup(Version,Details) to locate a Windows DLL
   WSAStartup has returned ImplementationDetails:
     wVersion:   (257)
     wHighVersion: (257)
     szDescription:'Microsoft Windows Sockets Version 1.1.'
```

```
    szSytemStatus:'Running.'
    iMaxSockets: (32767)
    iMaxUdpDg:  (65467)
    lpVendorInfo (00000010)

   Calling socket(AddressFormat,Type,Protocol) to Create a Client Socket
   Client Socket (56) has been created

   Calling connect(ServerSocket,AddressPointer,length)
   Connect succeeded: A Server is available

Enter data to transmit to the server
Client> {28r}C5:M1
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
   11 characters sent to the Server
   Data: {28r}C5:M1
Client> more data
Calling ioctlsocket(SOCKET,8004667e,->[1]) to set non blocking mode
ioctlsocket succeeded

   Calling send(Socket,buffer,SizeOfBuffer,Flags)
Error: (WSAECONNRESET      )

( Вызов WSAStartup(Version,Details) для определения места Windows DLL
WSAStartup возвратила детали реализации:
wVersion: (257)
wHighVersion: (257)
szDescription: 'Microsoft Windows Sockets Version 1.1'
szSystemStatus: 'Running'
iMaxSockets: (32767)
iMaxUdpDg: (65467)
lpVendorInfo (00000010)

Вызов гнезда(AddressFormat,Type,Protocol) для создания гнезда Клиента
Гнездо Клиента (56) создано

Вызов connect(ServsrSocket,AddressPointer,length)
Соединение прошло: сервер доступен

Ввести данные для передачи на сервер
Клиент>{28r}C5:M1
Вызов ioctlsocket(SOCKET,8004667e,->[1] для установки неблокирующего режима
ioctlsocket прошла

   Вызов send(Socket,buffer,SizeOfBuffer,Flags)
   11 знаков послано на сервер
   Данные: {28r}C5:M1
```

```
Клиент> новые данные
Вызов ioctlsocket(SOCKET,8004667e,->[1]) для установки неблокирующего режима
ioctlsocket прошла

Вызов send(Socket,buffer,SizeOfBuffer,Flags)
Ошибка: (WSAECONNRESET )
```

Глава 6

Библиотека гнезд C++

ВВЕДЕНИЕ

Цель

Основной целью настоящего примера является показ того, как следует реализовывать полную библиотеку классов C++, которая позволит вам манипулировать сетевыми гнездами. Эта библиотека классов обеспечит всю поддержку, которая "требуется" для нормального класса и предоставит все дополнительные компонентные функции для обращения с гнездами. Используемая модель ввода/вывода позволит вам применить в классе стандартные операторы вставления (<<) и изъятия (>>) в C++ для отсылки данных в сеть и получения данных из сети. Помните, что эти операторы используются с функцией C++ *cout*, которая занимает место *printf* и использует файл включения IOSTREAM.H. Пример:

```
cout << "IPC programming is fun!\n";
```

Имейте в виду, что если у вас недостаточно опыта в работе с построением класса C++, вам следует посмотреть комплект документации по Microsoft Visual C++ и пройти соответствующий курс обучения.

Класс гнезд

Библиотека классов, описываемая в этом примере, является классом **CSocket**. Класс гнезд будет использоваться следующим образом:

```
Csocket s;                          //создание гнезда сервера
CSocket c;                          //создание гнезда клиента
s.bind(100)                         //привязка к порту 100
s.listen(3)                         //прослушивание 3 клиентов
c = s.accept                        //принятие клиента
c << "Message sent to client";      //отсылка к клиенту
c >> buffer                         //прием от клиента
```

Ортодоксальная каноническая форма

Нетривиальные классы C++ должны иметь следующие четыре типа функций, для того чтобы считаться полными:

- Default Constructor (конструктор по умолчанию) CSocket::CSocket()

  Эта функция используется для создания нового экземпляра переменной гнезда класса с использованием значений создания по умолчанию.

- Copy Constructor (конструктор копирования) CSocket::CSocket(const socket&)

  Эта функция используется для создания нового экземпляра переменной гнезда класса с помощью существующего гнезда в качестве образца для создания нового гнезда.

- Destructor (деструктор) CSocket::~CSocket()

  Эта функция используется для очистки данных, когда переменная выходит из области действия.

- Assignment Operator (оператор присваивания)

  CSocket&CSocket::operator=(const CSocket&)

  Эта функция вызывается, когда вы имеете логику, которая "приписывает" одно гнездо к другому.

Примеры:

Нижеприведенный код показывает примеры четырех вышеупомянутых функций:

int MyFunction(Socket);	Определение функции MyFunction, которая принимает переменную типового гнезда.
{	
CSocket s;	Конструктор по умолчанию, вызванный для создания s.
CSocket *c;	Указатель на объявленное гнездо.
c = new CSocket	Конструктор по умолчанию, вызванный для создания гнезда, на которое указывает C.
CSocket y(s)	Конструктор копирования вызывается для создания новой переменной y из существующей s.
CSocket z=s;	Конструктор копирования вызывается для создания новой переменной z из существующей s.
MyFunction(s);	Конструктор копирования вызывается для создания временной переменной для использования в функции MyFunction. Первоначальная переменная s не передается в функцию, потому что s передается **по значению, а не по обращению.**

CSocket q;	Конструктор по умолчанию, вызванный для создания q.
q = s;	Оператор присваивания, вызванный для переназначения значения q.
delete c;	Деструктор вызывается для освобождения всех ресурсов, связанных с гнездом, на которое указывает
c.	
}	Деструктор вызывается для q, z, y и s, так как они выходят из области действия.

Модель ввода/вывода C++

C++ использует модель ввода/вывода, основанную на классах, производных от **iostream**. Базовый класс iostream используется для обеспечения характерного ввода и вывода для стандартных, заранее определенных типов C и C++: int, float, char, и т.д. Класс iostream имеет связанный с ним класс **streambuf**, который отвечает за реальное осуществление буферирования на физическое устройство и с него. iostream обеспечивает компонентные функции ввода/вывода и манипуляторы для каждой операции, и он, в свою очередь, вызывает **streambuf** для осуществления реального ввода/вывода.

Класс iostream реализует ввод/вывод *переопределением* (*overloading*) операторов >> и << для работы с конкретным классом. Переопределение оператора означает, что оператор, такой как <<, переопределяется для обеспечения мультифункциональности. Когда реализован класс, определяется "дружественная" функция. Дружественная функция будет иметь доступ к внутренним деталям класса. Она реализует ввод/вывод для класса с помощью стандартного ввода/вывода для iostream и заранее определенных типов C и C++.

Для того чтобы класс **CSockets** вел себя соответствующим образом, вы должны определить новый класс **streambuf**, который будет переопределять свой ввод/вывод в сеть, и вы должны присоединить этот **streambuf** к классу **CSocket**. Сам класс **CSocket** производится от iostream, поэтому он наследует все возможности iostream из его базового класса. С помощью подключения нового **streambuf** к нашему классу **CSocket** вы переопределяете в сети все вводы/выводы, направленные к классу **CSocket**.

CSOCKET.H

Введение

В любую программу, которая хочет использовать класс **CSocket**, который вы создаете, включается заголовочный файл **socket.h**. Заголовочный файл определяет различные классы и компонентные функции, которые используются в классе **CSocket**.

CSOCKET.H С ПОЯСНЕНИЯМИ

```
#ifdef WIN
#include <windows.h>
#else extern "C"
{
#include <prgpre.h>
}
#endif
#include <iostream.h>
#include "portable.h"
```

Эта часть программы предназначена для того, чтобы тот же самый исходный файл мог быть использован с MS-DOS и Windows. У клиента Windows NT и Windows 3.1 вы определяете макрокоманду WIN для включения windows.h. В платформе MS-DOS вы не определяете макрокоманду WIN, что означает, что включается заголовочный файл PATHWORKS prgpre.h. Заголовочный файл portable.h включает в себя дополнительные определения, которые нужны для того, чтобы сделать функции портативными.

```
typedef char *INADDR_CHAR;
```

Эта строка используется для определения нового типа, INADDR_CHAR, который является последовательностью (string) знаков, представляющих межсетевой адрес. Для принятия INADDR_CHAR в качестве параметра некоторые из функций **CSocket** переопределяются. Таким образом, вам нужно определить этот тип.

```
class CSocket;
```

Класс **CSocket** имеет обращение (reference) от класса **CSocketBuf**, а класс **CSocketBuf** имеет обращение от **CSocket**. Один из этих двух классов должен быть определен заранее. Так как обращение в классе **CSocketBuf** является *указателем* (*pointer*) на класс **CSocket**, а обращение в классе **CSocket** является *встроенным* (*embedded*) классом **CSocketBuf**, класс **CSocket** является классом, который может быть заранее определен. Вы не будете знать все детали, относящиеся к **CSocket**, когда мы определяем **CSocketBuf**, но так как элемент в **CSocketBuf** является указателем, вы знаете об указателе достаточно, для того чтобы построить **CSocketBuf**. В принципе, вам нужно знать размер указателя, который всегда один и тот же.

```
class CSocketBuf : public streambuf
//CSocketBuf is derived from StreamBuf
{ public:
  CSocket * SocketPointer;
```

```
  CSocketBuf();
  virtual int doallocate();
  virtual int overflow(int ch);
  virtual int underflow();
  virtual int sync();
  void setBuf(CSocket *);
};
```

Для класса **CSocketBuf** объявляется новая компонентная переменная (memder variable): *SocketPointer*. Эта переменная является указателем на **CSocket**, который будет использован в качестве физического устройства для этого **streambuf**. Данные будут отсылаться и считываться с этого **CSocket**, когда это будет необходимо.

При порождении нового класса из **streambuf** требуется, чтобы вы обеспечили три функции, и вам разрешается переопределять дополнительные функции. Этот код обеспечивает реализации для следующих трех требуемых функций:

- *doallocate()*

Эта функция вызывается со стороны **streambuf** для первоначального распределения буфера с целью хранения местных (локальных) данных. Когда данные вводятся и выводятся из соответствующего iostream, он будет вызывать **streambuf** для отсылки и считывания данных с физического устройства.

- *overflow(int ch)*

Когда iostream пытается вставить данные в **streambuf**, а буфер уже полон, логика **streambuf** автоматически вызывает эту функцию, вместе с тем знаком, который пытались ввести. Целью этой функции является отсылка на физическое устройство информации, которая находится в части выходного буфера, относящейся к **streambuf**.

- *underflow()*

Когда iostream пытается извлечь данные из **streambuf**, а буфер пуст, логика **streambuf** автоматически вызывает эту функцию. Целью этой функции является считывание информации с физического устройства и заполнение части входного буфера, относящейся к **streambuf**.

Вам разрешается переопределить следующую функцию, хотя вы и не обязаны это делать:

- *sync()*

Когда iostream пытается сбросить данные в **streambuf**, он вызывает эту функцию. Эта функция сбрасывает все данные на физическое устройство и освобождает входной и выходной буферы.

Следующие функции являются новыми функциями в классе **CSocketBuf**:

- *CSocketBuf()* Конструктор по умолчанию;
- *SetBuf(CSocket*)* Используется для установки значения для переменной *SocketPointer*.

```
class CSocket : public iostream
{
 public:
 static int flagdata;          //используется для установки начального состояния флагов
```

Компоненты общедоступных данных

Целочисленная переменная *flagdata* используется для установления первоначальных значений флаговых компонент **CSocket**. Так как эта переменная объявляется как **статическая** компонента, существует только один экземпляр этой переменной для всего класса, и он разделяется между всеми гнездами **CSocket**, которые вы создаете. Это разрешено, так как единственный раз, когда эта переменная используется, — это во время создания нового **CSocket**. Основной причиной этого является возможность задействования регистрации (logging) во время самого процесса построения.

Для того чтобы задействовать регистрацию в фазе построения, вы устанавливаете переменную *flagdata*, так что регистрация задействована. Во время построения переменная *flags* будет установлена надлежащим образом, и регистрация осуществится автоматически. Эта переменная должна быть общедоступной (public), поэтому вы можете устанавливать ее вне экземпляра **CSocket**.

```
 private:
  friend class  CSocketBuf;
  static int count;
CSocketBuf     sBuf;
int            flags;
int            status;
int            s;
SOCKADDR_IN  *  Address;
SOCKADDR_IN  *  LastAddress;
```

Компоненты закрытых данных

Следующие переменные объявляются как компоненты закрытых (частных) данных. Это означает, что единственным способом получить доступ к переменным является доступ через компонентную функцию **CSocket**, или же при наличии дружественной функции (a friend) по отношению к классу **CSocket**. Отметим, что весь класс **CSocketBuf**

объявлен дружественным по отношению к классу **CSocket**. Любая компонентная функция класса **CSocketBuf** может иметь доступ к закрытым частям класса **CSocket**.

- *static int count*

Это — статическая переменная, поэтому существует только один экземпляр этой переменной для всего класса. Она используется для подсчета количества реализованных **CSockets**.

- *CSocketBuf sbuf*

Эта переменная представляет собой *встроенный* **CSocketBuf**, который используется как **streambuf** для отсылки и считывания данных с физического устройства. В этом случае это устройство представляет собой сеть.

- *int flags*

Эта переменная используется для отслеживания всех флагов, которые могут быть установлены.

- *int status*

Эта переменная применяется для хранения в памяти статуса последней гнездовой операции, если это необходимо.

- *int s*

Это — переменная, которая содержит гнездо, возвращенное от функции *socket()*.

- *SOCKADDR_IN*Address*

Эта переменная содержит адрес, используемый для *привязывания* (*bind*) или *присоединения* (*connect*).

- *SOCKADDR_IN *LastAddress*

Эта переменная содержит последний адрес, который использовался для связи без логического соединения (connectionless-based communications).

```
public: CSocket();                          // Конструктор по умолчанию
  CSocket(int);                             // Построение с существующим ранее гнездом
  CSocket(CSocket&);                        // Конструктор копирования
  ~CSocket();                               // Деструктор
  CSocket & operator = (CSocket &);         // Оператор присваивания
```

Ортодоксальные канонические функции

Следующие функции являются стандартными функциями, которые должны обеспечиваться для реализации ортодоксального класса:

- Normal Constructors (нормальные конструкторы)

  Объявляется, что конструктор по умолчанию *CSocket()* создает новое гнездо по умолчанию **CSocket**. Объявляется, что специальный конструктор *CSocket(int)* создает новое **CSocket**, когда сетевое гнездо — int — уже существует.

- Copy Constructor (конструктор копирования)

  Конструктор копирования используется для создания нового **CSocket**, когда существующее **CSocket** используется в качестве шаблона для нового гнезда. Обычно в конструкторе копирования компоненты нового элемента (new item) копируются из существующего элемента.

- Destructor (деструктор)

  Функция деструктора используется для закрывания сетевого гнезда s и освобождения всех буферов, размещенных в **CSocket**. Счетная компонента статических данных также уменьшается на единицу для указания того, что стало одним гнездом **CSocket** меньше.

- Assignment Operator (оператор присваивания)

  Оператор присваивания используется, когда вы имеете две действующих переменных **CSocket**, и вы "приписываете" (assign) одну переменную другой. Это аналогично случаю конструктора копирования, но главным отличием является тот факт, что цель присваивания уже существует. В операторе присваивания вы должны выполнить шаги, требуемые для удаления цели, и затем осуществить переназначение его компонент. Эта функция должна разрушить цель и должна заново инициализировать данные цели для того, чтобы они были такими же, как и исходные данные.

```
BOOL isLogging();
BOOL isBound();
BOOL isListening();
BOOL isConnected();
```

Флаговые функции

Для определения состояния **CSocket** используются различные флаговые функции. Так как компонента флаговых данных является компонентой закрытых данных, вы не можете осуществить доступ к флаговой переменной вне компонентной функции класса **CSocket**. Эти функции позволяют вам получить доступ к соответствующей части флаговой переменной для проверки того, установлено ли соответствующее состояние или нет. Имеются следующие функции:

- *isLogging()* — указывает, задействована ли регистрация (logging);
- *isBound()* — указывает, привязано ли CSocket к сетевому адресу;

- *isListening()* — указывает, осуществляет ли **CSocket** прослушивание в настоящий момент;
- *isConnected()* — указывает, соединено ли в настоящий момент **CSocket** с другим гнездом.

```
int          getSocketValue() {return(s);};
virtual int   error(char *, ...);
int          setLogging(int);
```

Функции обращения с закрытыми данными

Эти три функции позволяют вам обращаться с различными компонентами закрытых данных **CSocket**:

- *getSocketValue*
  Эта функция возвращает текущее значение сетевого гнезда **s**. Эта функция позволяет вам считать значение s, однако соответствущего метода установления значения нет.
- *error(char*,...)*
  Эта функция предназначена для повторной реализации, что случается, когда в классе, порожденном от класса **CSocket**, возникает ошибка.
- *setLogging(int)*
  Эта функция позволяет вам установить значение для флага регистрации.

```
int          bind(unsigned short,INADDR_CHAR);
int          unbind();
int          listen(int);
int          StopListening();
CSocket &     accept();
int          connect(unsigned short,INADDR_CHAR);
int          disconnect();
```

Гнездовые функции

Эти функции объявляют стандартные гнездовые функции из класса **CSocket**. Каждая функциональная компонента требует только тех аргументов, которых нет внутри компонент данных класса **CSocket**:

- *bind(...)*
  Эта функция позволяет вам определить порт и адрес для привязывания к гнезду. Адрес вводится в виде последовательности знаков, поэтому вы не должны распределять и заполнять члены структуры SOCKADDR_IN. Аргумент адреса будет устанавливаться в подходящее значение, если он не поставляется.

- *unbind(...)*

  Эта функция предназначена для "отвязывания" гнезда от адреса, если оно в настоящее время уже привязано. Реализация гнезд Berkeley не дает метода "развязывания", поэтому эта функция в той платформе не реализуется.

- *listen(int)*

  Эта функция используется для установления очереди на прослушивание для клиентов, которые могут подключаться и приниматься сервером.

- *connect(...)*

  Эта функция позволяет вам подсоединяться к серверу. Вы можете определить порт и адрес сервера.

- *disconnect()*

  Эта функция предназначена для отсоединения от гнезда управляемым способом.

```
int        recvfrom(char  *,int,int=0);
int        sendto(char  *,int,int=0,INADDR_CHAR="",unsigned short=0);
int        recv(char  *,int,int=0);
int        send(char  *,int,int=0);
};
```

Функции передачи данных

Эти функции могут быть вызваны, если вы хотите отослать данные на сетевое гнездо без использования операторов C++ вставки (<<) или изъятия (>>). Каждая функция ведет себя аналогично соответствующей гнездовой функции.

CSOCKET.CPP (С ПОЯСНЕНИЯМИ)

```
#include "CSOCKET.H"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#ifndef WIN
#include <dneterr.h>
#endif
```

Включаются следующие заголовочные файлы:

- CSOCKET.H — объявляет класс **CSocket** и компонентные функции;
- stdio.h — объявляет стандартные функции ввода/вывода;

- string.h — объявляет функции обращения с последовательностью;
- stdlib.h — объявляет другие функции, нужные для этого примера;
- dneterr.h — поставляется сервисом распространения PATHWORKS; объявляет ошибочные числа, которые нужны только при программировании на уровне MS-DOS.

```
void startnet()
 {
#ifdef WINNT
  WORD    VersionRequested;
  WSADATA   ImplementationDetails;
  VersionRequested = MAKEWORD(1,1);
  int rv = WSAStartup (VersionRequested, &ImplementationDetails);
  if (0 != rv )
   {
    cout << "Sorry, can not start the Network" << endl;
   }
#endif
 }

void stopnet()
 {
 }
```

Эти две глобальные функции предусмотрены для инициализации сети и для закрытия сети. *Startnet()* автоматически вызывается, когда создается первое **CSocket**, а *stopnet()* вызывается, когда ликвидируется последнее гнездо **CSocket**.

```
int CSocket::count=0;
int CSocket::flagdata=0;
```

Две статические компоненты класса **CSocket** объявляются в заголовочном файле **CSocket.h**, но они должны быть один раз определены в коде, для того чтобы зарезервировать место в памяти для переменной. Это место в коде программы, где определяются переменные.

```
void output(char *buffer, char *end)
 {
 int count = end - buffer;
 while (count > 0)
  {
   if (31>*buffer) printf("\\%03d",*buffer);
   else     printf("%c",*buffer);
   buffer++;
   count—;
  }
```

```
 printf("\n");
 }

void output(char *buffer, int count)
 {
 while (count > 0)
  {
   if (31>*buffer) printf("\\%03d",*buffer);
   else      printf("%c",*buffer);

   buffer++;
   count—;
  }
 printf("\n");
 }
```

Существуют две версии функции *output* (вывод) в библиотеке классов **CSocket**. Первая версия принимает два указателя: указатель на начало последовательности знаков и указатель на конец последовательности знаков. Эта функция вывода выведет на печать все знаки между этими двумя указателями.

Вторая версия функции output принимает указатель на последовательность знаков и счет количества знаков в последовательности. Эта функция вывода выводит на печать соответствующее количество знаков.

Это является хорошим примером **переопределения функции (function overloading)**, где одно и то же имя функции может быть использовано с различными аргументами. Соответствующая функция, подлежащая вызову, определяется проверкой аргументов и согласованием их с соответствующей функцией. Отметим, что действительная функция делает то же самое — она просто имеет два способа задания аргументов.

```
CSocketBuf::CSocketBuf()
 { unbuffered(0);
  allocate();
 }
```

Это конструктор для **CSocketBuf**. **CSocketBuf** представляет собой место, где **CSocket** будет буферировать данные, перед тем как они будут посланы/приняты из сети. Следует помнить, что **CSocketBuf** порожден базовым классом **streambuf**. Конструктор вызывает две функции в базовом классе для инициализации **streambuf**, который встроен в **CSocketBuf**. *Небуферируемый* (*unbuffered(0)*) вызов указывает на то, что **streambuf** не собирается разбуферироваться, т.е. **streambuf** представляет собой буферируемую версию. Функция *allocate()* вызывается для распределения памяти для буфера, а также автоматически вызывает виртуаль-

ную функцию *doallocate()* для установки распределения буфера. Функция *doallocate()* является виртуальной функцией для того, чтобы **CSocketBuf** могла переопределить реализацию, которая была обеспечена в классе **streambuf**.

```
CSocketBuf::doallocate()
 {
  int i = blen()+1000;
  char * p = new char [i];
  setb(p,p+i,1);
  setp(p+(i/2)+1,p+i);
  setg(p,p+(i/2)-1,p+(i/2)-1);
  return(1);
}
```

Это функция, которая распределяет буфер и подразделяет его на буфер ввода и буфер вывода. Используются следующие компонентные функции **streambuf**:

- *blen()* — возвращает текущую длину буфера — должна быть 0;
- *setb(...)* — устанавливает значения для базового указателя (bp) и указателя на конец (ebuf);
- *setp(...)* — устанавливает значения для базы зоны Put (pbase) и конца зоны Put (eppt);
- *setg(...)* — устанавливает значения для начала зоны получения (eback), зоны следующего получения (gtpr) и конца зоны получения (egptr).

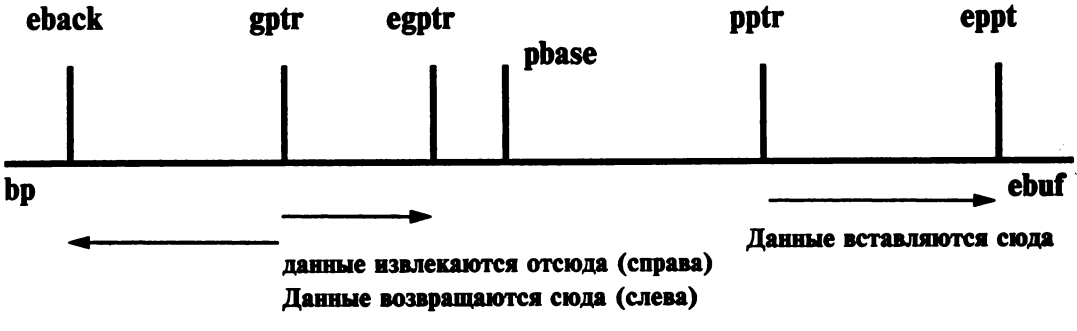


Рис. 6-1. Компонентные функции streambuf.

```
inline void CSocketBuf::setBuf(CSocket *in)
 {
  SocketPointer = in;
 }
```

Это новая компонентная функция, которая предназначена для установки значения компоненты закрытых данных SocketPointer. Ожидается, что **CSocket** вызовет эту функцию для указания себе самой на свою встроенную функцию **CSocketBuf**.

```
int CSocketBuf::overflow(int ch)
 {
  if (SocketPointer->isLogging()) output(pbase(),pptr());
  if (SocketPointer->isConnected())
   {
    if (SocketPointer->isLogging())
          cout << "CSocketBuf:Sending" << endl;
    SocketPointer->send(pbase(),pptr()-pbase());
   }
 else
   {
    if (SocketPointer->isBound())
         {
          if (SocketPointer->isLogging())
             cout << "CSocketBuf:SendTo(" <<
                    inet_ntoa(SocketPointer->LastAddress->sin_addr) <<"port" <<
                    SocketPointer->LastAddress->sin_port <<endl;

          SocketPointer->sendto(pbase(),pptr()-pbase());
         }
    else if (SocketPointer->isLogging())
     cout << "This socket is not bound and is not connected, cannot send" << endl;
   }
 setp(pbase(),epptr());
 if (EOF!=ch) sputc(ch);
 return(1);
}
```

Эта функция вызывается, когда **CSocket** пытается вставить данные в **CSocketBuf**, а зона Put — между *pbase* и *eppt* — заполнена. В этом случае вы должны послать данные в зоне put в сеть и затем освободить зону put. Вы должны решить, что делать со знаком, который вызвал состояние переполнения, т.е. вы должны решить, посылаете ли его тоже, или помещаете его в зону put, для того чтобы отослать его в следующий раз.

Чтобы освободить зону put, вам необходимо сделать следующее:

- Если **CSocket**, связанное с этим **CSocketBuf** [SocketPointer], осуществляет регистрацию, тогда нужно осуществить вывод (*output*) зоны put.
- Отослать данные в сеть. — Если **CSocket** соединено с другим гнездом сети, тогда выдайте команду send на отсылку данных

в зоне put в сеть. Если **CSocket** не имеет такого соединения, но имеет привязку, тогда отошлите зону put в *LastAddress* (последний адрес), из которого вы получили данные, т.е. будет имеет место передача без логического соединения. Если **CSocket** не имеет ни соединения, ни привязки, тогда выдайте на печать состояние ошибки.

- После отсылки данных сбросьте *pptr* обратно в начало зоны put и вставьте знак, который вызвал переполнение, если только он не является знаком EOF.

```
int CSocketBuf::underflow()
 {
  char buffer[121];
  int l;
  if (gptr()!=egptr()) return(*gptr());
  if (SocketPointer->isConnected())
    {
     if (SocketPointer->isLogging())
       cout << "CSocketBuf:Recving" << endl;
     l = SocketPointer->recv(buffer,120);
    }
  else
    {
     if (SocketPointer->isBound())
         {
           if (SocketPointer->isLogging())
             cout << "CSocketBuf:recvfrom called" << endl;
           l =SocketPointer->recvfrom(buffer,120);
         }
     else
         {if (SocketPointer->isLogging())
              cout << "This socket is not bound and is not connected"
                   << ", can not recvfrom" << endl;
          l=0;
         }
    }
 if (l<=0)
   {
    // if no data, provide at least NUL data for input....
    // Otherwise infinite loop....
    buffer[0]=32;
         buffer[1]=0;
    l=1;
 }
 buffer[l]=0;
 l = strlen(buffer);
 strncpy(egptr()-l,buffer,l);
 setg(eback(),egptr()-l,egptr());
 return(*gptr());
}
```

Эта функция вызывается, когда **CSocket** пытается извлечь данные из **CSocketBuf** и зона get между *eback* и *egptr* пуста. В этом случае вы должны запросить выдачу данных сети для заполнения зоны get.

Логика аналогична функции переполнения (Overflow):

- проверить и удостовериться, что вы осуществляете регистрацию;
- принять данные (recv):
  - — от подсоединенного Клиента,
  - — от любого клиента, использующего *recvfrom* с передачей без установления логической связи,
  - — вывести на печать ошибку;
- скопировать данные в зону Get, выстроенные таким образом, что концом данных является *egptr*.

```
int CSocketBuf::sync()
 {
 overflow(EOF);
  return(0);
 }
```

Это — функция, которая вызывается для сброса всех данных из зоны put. Для сброса данных вы используете функцию переполнения (overflow).

```
CSocket::CSocket() : iostream(& sBuf)
 {
  flags     =flagdata;
  status    =0;
  Address   = new SOCKADDR_IN;
  LastAddress = new SOCKADDR_IN;
  sBuf.setBuf(this);

  if (0==count)
   {
     startnet();
   }
  s = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
  if (SOCKET_ERROR == s) error("Creating Socket");
  else
   {
    count++;
    if (isLogging())
        cout << "Created socket (" << s << ") total sockets:" << count << endl ;
   }
 }
```

Это — конструктор по умолчанию для **CSocket**. Отметьте, что компонентные флаги (*flags*) закрытых данных инициализируются от открытых статических компонентных флаговых данных (flagdata). К тому времени, когда вы доберетесь до этой функции, **CSocketBuf** уже полностью создан и его конструктор вызван. Оператор *sBuf.setBuf(this)* связывает это **CSocket** с его встроенным **CSocketBuf**. Элемент закрытых данных *SocketPointer* буфера **CSocketBuf** будет установлен для указания на создаваемое в текущий момент **CSocket**.

Если текущий счет гнезд **CSockets** равен 0, тогда вам нужно запускать сеть.

Функция *socket* используется для создания сетевого гнезда, связанного с этим **CSocket**, и ее значение хранится в компоненте *s* закрытых данных.

Если гнездо создано успешно, тогда счет гнезд **CSockets** увеличивается на единицу и на печать выводится сообщение регистрации (log message), если на этом **CSocket** задействована регистрация. Если это так, то она должна была быть задействована во флаговых данных (*flagdata*).

```
CSocket::CSocket(int sock) : iostream(& sBuf)
 {
  flags     =flagdata;
  status    =0;
  Address     = new SOCKADDR_IN;
  LastAddress  = new SOCKADDR_IN;
  sBuf.setBuf(this);
  s = sock;

  if (SOCKET_ERROR==sock)
   {
     if (isLogging())
       cout << "Error Socket Initialized" << endl;
     return;
   }

 if (0==count)
  {
   startnet();
  }
 count++;
 flags = flags | 0x05;
 if (isLogging())
   cout << "Initialized socket (";
   cout << s;
   cout << ") total sockets:";
   cout << count;
   cout << endl ;
 }
```

Это — альтернативный конструктор для **CSocket**. Если сетевое гнездо уже существует, тогда вы создаете **CSocket**, которое уже имеет компоненту данных *s*, установленную на существующее гнездо вызовом этого конструктора. Единственным параметром, переданным в конструктор, является номер гнезда для существующего гнезда. Если вы хотите создать **CSocket**, однако не хотите иметь его сразу же привязанным к сетевому гнезду, вы можете передать ему значение SOCKET_ERROR и оно будет создано как гнездо ошибки.

```
CSocket::CSocket(CSocket &in)
 {
  flags     =flagdata;
  status    =0;
  Address     = new SOCKADDR_IN;
  LastAddress  = new SOCKADDR_IN;
  sBuf.setBuf(this);
  s = in.GetSocketValue();

  if (SOCKET_ERROR==sock)
   {
    if (isLogging())
      cout << "Error Socket Initialized" << endl;
    return;
   }

 count++;
 flags = flags | 0x05;
 if (isLogging())
   cout << "Initialized socket (";
   cout << s;
   cout << ") total sockets:";
   cout << count;
   cout << endl ;
 }
```

Это — конструктор копирования для **CSocket**. Так как сетевое гнездо уже существует, вы создаете новое **CSocket**, которое уже имеет компоненту данных *s*, установленную в существующее гнездо с помощью вызова этого конструктора. Единственным параметром, передаваемым в конструктор, является существующее **CSocket**, которое вы используете для выяснения, какой номер гнезда следует использовать для существующего сетевого гнезда.

```
CSocket::~CSocket()
 {
  if (SOCKET_ERROR != s)
   {
   cout << "Flushing socket (" << s << ")" << endl;
   sync();
   }
```

```
 delete Address;
 delete LastAddress;

 if (SOCKET_ERROR == s)
  {
   if (isLogging())
      cout << "Error Socket closed" << endl;
   return;
  }
 count— —;
 if (isLogging())
  cout << "Closing socket(" << s << ") " << endl;

 int rv = sclose(s);
 if (-1 == rv) error("closing socket");

 if (0==count)
  {
   stopnet();
  }
 if (isLogging())
  cout << "Deleted socket(" << s << ") total sockets:" << count << endl;
}
```

Это — деструктор для **CSocket**. Его целью является очистка всех данных, связанных с **CSocket**, до разрушения **CSocket**. Данные и ситуации, которые нуждаются в очистке, включают в себя:

- Любые данные в соответствующем **CSocketBuf**, которое нужно разгрузить.
- *Address* и *LastAddress*, которые являются указателями на структуры, распределенные при построении. При разрушении эта память вычеркивается.
- Сетевое гнездо нужно закрыть, если оно открыто.
- Если это последнее **CSocket**, то сеть нужно закрыть.

```
CSocket & CSocket::operator = (CSocket &right)
 {
  if (SOCKET_ERROR != s)
   {
   cout << "Flushing socket (" << s << ")" << endl;
   sync();
   }
  if (SOCKET_ERROR == s)
   {
    if (isLogging())
     cout << "Error Socket closed" << endl;
   }
```

```
 else
  {
   if (isLogging())
   cout << "Closing socket(" << s << ") " << endl;

   int rv = t_close(s);
   if (-1 == rv) error("closing socket");
     }
  if (isLogging())
    cout << "Socket (" << s << ") being replaced by (" << right.s << ")" << endl;

  s = right.s;
  flags = right.flags;
  return(*this);
}
```

Это — оператор присваивания (assignment operator) для **CSocket**. Когда вы присваиваете **CSocket** другому **CSocket**, цель вычеркивается и исходный текст возвращается как значение для нового гнезда.

```
BOOL CSocket::isBound()
 {
  return(flags & 0x01);
 }
BOOL CSocket::isListening()
 {
  return(flags & 0x02);
 }
BOOL CSocket::isConnected()
 {
  return(flags & 0x04);
 }
BOOL CSocket::isLogging()
 {
  return(flags & 0x10);
 }
```

Эти функции позволяют вам проверить различные биты переменной *flags*, чтобы посмотреть, установлен ли бит. Флаговые биты устанавливаются различными компонентными функциями, которые осуществляют привязку, соединение или прослушивание гнезда. Для установления бита регистрации (log bit) используется следующая функция:

```
int CSocket::setLogging(int flag=1)
 {
  if (1==flag)
```

```
    flags = flags | 0x10;          // set the bit on
  else flags = flags & 0xFEFF;     // mask the bit off
  return(0);
 }
int CSocket::bind(unsigned short port, INADDR_CHAR addr)
 {
  if (SOCKET_ERROR == s)
  { if (isLogging())
      cout << "can not Bind an error socket" << endl;
return(-1);

   }
  if (isBound())
  { if (isLogging())
     cout << "Socket " << s << " Already Bound" << endl;
     return(-1);
  }
  Address->sin_family      = AF_INET;
  Address->sin_port        = port;
  Address->sin_addr.s_addr = inet_addr(addr);

  int rv = ::bind(s,(LPSOCKADDR)Address,sizeof(*Address));
  if (-1 == rv) return(error("Binding socket"));
  else flags = flags | 0x01;

  if (isLogging())
    cout << "Binding(" << s << ") to " << addr << " port " << port << endl; return(0);
}
```

Функция *bind* используется для привязывания существующего гнезда *s* — созданного, когда создавалось **CSocket** — к сетевому адресу и порту. Если значение гнезда недействительно или уже привязано к какому-либо адресу, на печать выводится сообщение ошибки. Отметьте, что оператор разрешения обзора (scope resolution operator) :: должен был использоваться для вызова функции *bind* WINSOCK. В противном случае функция *bind* вызвала бы сама себя и возникла бы бесконечная рекурсия. После успешного осуществления привязывания переменная *flags* устанавливается для индикации того, что *CSocket* привязано к адресу, а компонента данных *Address* указывает адрес, к которому оно привязано.

```
int CSocket::unbind()
 {
  if (isLogging())
   cout << "Unbind Not supported in TCP/IP" << endl;
  return(-1);
 }
```

Функция *unbind()* не поддерживается в гнездах TCP/IP.

```
int CSocket::listen(int queue)

 {
  if (SOCKET_ERROR == s)
   { if (isLogging())
    cout << "can not Listen on an error socket" << endl;
    return(-1);

   }
  if (isListening())
   { if (isLogging())
       cout << "Socket " << s << " Already Listening" << endl;
       return(-1);
   }
  int rv = ::listen(s,queue);
  if (-1 == rv) return(error("Listening..."));
  else flags = flags | 0x02;

  if (isLogging())
    cout << "socket " << s << " is now listening" << endl;
    return(0);
 }
```

Функция *listen* вызывается для установления очереди на прослушивание для гнезда. Единственный параметр указывает, сколько клиентов могут присоединяться без их принятия. Если сетевое гнездо *s* является недействительным гнездом или оно уже прослушивается, то на печать выводится сообщение об ошибке.

Заметим, что для получения версии WINSOCK функции *listen* должен использоваться оператор разрешения обзора, так что ваша функция *listen* сама себя не вызывает. Для указания того, что гнездо слушает, устанавливается переменная flags (флаги).

```
int CSocket::StopListening()
 {
  if (isLogging())
    cout << "Stop Listening Not supported in TCP/IP" << endl;
  return(-1);
 }
```

В TCP/IP функциональность *StopListening* (*окончание прослушивания*) не поддерживается.

```
CSocket & CSocket::accept()
 {
  CSocket *cs;
```

```
 int size=sizeof(SOCKADDR_IN);
 if (SOCKET_ERROR == s)
 { if (isLogging())
    cout << "can not accept on an error socket" << endl;
  return *this;
 }
if (isLogging())
 cout << "Calling accept to get a client, this may block" << endl;

int sock = ::accept(s,(LPSOCKADDR)LastAddress,&size);
if (SOCKET_ERROR == sock)
  {
    error("Accepting a client");
    cs = new CSocket(SOCKET_ERROR);
    return(*cs);

 }
if (isLogging())
  cout << "A new Client is available at " <<
    inet_ntoa(LastAddress->sin_addr) << " port " <<
    LastAddress->sin_port << endl << endl <<
   "Creating a [connected] client socket" << endl;

 cs = new CSocket(sock);
 return(*cs);
}
```

Для возвращения гнезда, которое может быть использовано для осуществления связи со следующим клиентом, который ожидает обработки, используется функция *accept*. Если гнездо является гнездом ошибки, тогда на печать выводится соответствующее сообщение.

Вызов функции *accept* может осуществлять блокировку, пока не появится клиент, ожидающий соединения.

Если во время действия *accept* произошла ошибка, тогда создается ошибочное **CSocket** для возвращения вызывающей программе (caller). В противном случае создается новое **CSocket**, которое подключается к существующему сетевому гнезду, которое было возвращено функцией *accept*.

Функция *accept* возвращает новое **CSocket** вызывающей программе.

```
int CSocket::connect(unsigned short port,INADDR_CHAR addr)
 {
  if (SOCKET_ERROR == s)
  { if (isLogging())
     cout << "cannot connect on an error socket" << endl;
   return(-1);
  }
```

```
 if (isConnected())
  {  if (isLogging())
       cout << "Disconnecting from existing connection first" << endl;
     disconnect();
  }

 Address->sin_family        = AF_INET;
 Address->sin_port = port;
 Address->sin_addr.s_addr     = inet_addr(addr);

 int rv = ::connect(s,(LPSOCKADDR)Address,sizeof(*Address));
 if (-1 == rv) return(error("connecting socket"));
 else flags = flags | 0x04;

 if (isLogging())
  cout << "Connecting(" << s << "to " << addr << " port " << port << endl;
 return(0);
}
```

Для подключения к существующему серверу используется функция *connect*. Аналогично функции *bind()* функция *connect* принимает порт и адрес сервера, с которым вы пытаетесь соединиться. Если вы пытаетесь соединиться и уже соединены с сервером, тогда существующее соединение отключается.

Переменная *flags* устанавливается для индикации того, что **CSocket** соединено.

```
int CSocket::disconnect()
{
 if (SOCKET_ERROR == s)
 { if (isLogging())
    cout << "can not disconnect an error socket" << endl;
   return(-1);
 }
 if (!isConnected())
  { if (isLogging())
     cout << "Not Connected" << endl;
    return(-1);
  }
 int rv = ::shutdown(s,2);
 if (-1 == rv) return(error("Shutdown....")); 
 else flags = flags & 0xFFFB;
 if (isLogging())
  cout << "Connection Terminated" << endl;
 return(0);
}
```

Эта функция будет отключать существующее соединение, если гнездо подключено. Она выполняет функцию WINSOCK *shutdown* для прекра-

щения соединения и затем изменяет переменную *flags* для указания нового статуса соединения.

```
int CSocket::recvfrom(char *buffer, int len, int flags)
 {
  if (isLogging())
     cout << "recvfrom not supported on DOS" << endl;
  return(-1);
 }

int CSocket::sendto(char * buffer, int len, int flags,INADDR_CHAR addr, unsigned short port)
 {
  if (isLogging())
    cout << "sendto not supported on DOS" << endl;
  return(-1);
 }
```

В этом примере функции *recvfrom* и *sendto* не поддерживаются, потому что DOS поддерживает только передачу данных, ориентированных на логическое соединение (connection-oriented data). Для поддержки этих функций вам следует вызвать функции WINSOCK *sendto* или *recvfrom*. В случае *recvfrom* вам нужно обновить элемент *LastAddress*, так чтобы следующая функция *sendto* могла использовать его в случае необходимости в качестве значения по умолчанию.

```
int CSocket::recv(char *buffer,int len, int flags)
 {
  if (SOCKET_ERROR == s)
  {
   if (isLogging())
     cout << "can not recv data over an error socket" << endl;
   return(-1);
  }
 if (!isConnected())
  {
   if (isLogging())
    cout << "can not recv data unless you are connected\nSocket:" << s << endl;
   return(-1);
  }
if (isLogging())
 cout << "Calling recv on socket (" << s << ")" << endl;
int rv = ::recv(s,buffer,len,flags);

 if (-1 == rv) error("receiving data ....");
 if (isLogging())
     {
      cout << "DATA:(" << s << ") len:" << rv << endl;
      output(buffer,rv);
    }
```

```
 return(rv);
}
```

Функция *recv* предусмотрена на случай, если вы захотите получить данные на сетевом гнезде без использования обычных операторов извлечения (extraction operators) языка C++. При желании вы можете непосредственно вызывать эти функции.

Если это ошибочное гнездо или если оно не соединено, тогда на печать будет выведено сообщение ошибки. В противном случае данные будут приниматься и, если задействована регистрация, они будут выводиться на печать.

```
int CSocket::send(char *buffer,int len, int flags)
 {
  if (SOCKET_ERROR == s)
  {
   if (isLogging())
    cout << "can not send data over an error socket" << endl;
   return(-1);
  }
   if (!isConnected())
  {
   if (isLogging())
    cout << "can not send data unless you are connected\nSocket:" << s << endl;
   return(-1);
  }
   if (isLogging())
    cout << "Calling send on socket (" << s << ")" << endl;
   int rv = ::send(s,buffer,len,flags);

   if (-1 == rv) error("send data ....");
   if (isLogging())
    {
    cout << "DATA:(" << s << ") len:" << len << endl;
    output(buffer,len);
    }
  return(rv);
 }
```

Эта функция предусмотрена на случай, если вы захотите послать данные через сетевое гнездо без использования обычных операторов вставки (insertion operators) языка C++. При желании вы можете непосредственно вызывать эти функции.

Если это ошибочное гнездо или если оно не соединено, тогда на печать будет выведено сообщение об ошибке. В противном случае данные будут передаваться и, если задействована регистрация, они будут выводиться на печать.

```
#ifdef WIN

int CSocket::error(char *data, ...)
 {
   char buffer[100];
   char *Msg;
   int err;
   err = WSAGetLastError();
   switch(err)
    {
      case(WSAEINTR        ): Msg="WSAEINTR        ";
        break;
      case(WSAEBADF        ): Msg="WSAEBADF        ";
        break;
      .
      .
      .

      case(WSANO_DATA      ): Msg="WSANO_DATA      ";
        break;
    default: sprintf(buffer,"Unknown: Error(%d)",err);
          Msg=buffer;
         break;

     }
    printf("error: (%s)\n",Msg);
    return(-1);
 }
```

Это процессор ошибки по умолчанию для класса **CSockets**. Он будет выводить на печать сообщение, указывающее, что имела место ошибка, и затем будет продолжать обработку. Этот раздел программы является специфичным для WINSOCK.

```
#else

int CSocket::error(char *data,...)
 {
  if (isLogging())
  {
   cout << "\n*****ERROR*****\n" << data << endl;
   cout << "SOCKET: (" << s << ")" << endl;
   cout << "Error Number: " << errno << endl;
   if (errno < SYS_NERR)
      cout << nsys_errlist[errno] << endl;
   else cout << "Unknown Error " << endl;
   cout << "****************" << endl;
  }
```

```
 else
  {
   cerr << "\n*****ERROR*****\n" << data << endl;
   cerr << "SOCKET: (" << s << ")" << endl;
   cerr << "Error Number: " << errno << endl;

   if (errno < SYS_NERR)
       cerr << nsys_errlist[errno] << endl;
   else cerr << "Unknown Error " << endl;
   cerr << "****************" << endl;
  }
  cerr << "Press return to continue" << endl;
  getchar();
  return(-1);
 }
#endif
```

Этот раздел программы предназначен для обработки ошибок на уровне MS-DOS с помощью библиотек PATHWORKS вместо WINSOCK.

ПРИМЕР СЕРВЕРА

```
#include "csocket.h"
#include <string.h>
#include <stdlib.h>

void output(char *, int);

int service(CSocket &);

main(int argc, char *argv[])
  {
    CSocket::flagdata=0x10;
    CSocket s;
    CSocket c;
    int port=100;
    if (2==argc) port = atoi(argv[1]);

    s.bind(port,"127.0.0.1");
    s.listen(3);
    int quit=0;
    while (quit<1)
    {
      c = s.accept();
      while(0==quit)
      {
      quit = service(c);
}
```

```
  }
 }

int service(CSocket & p)
 {
  char buffer[100];
  int i=-1,j=-1;
  char c='A';
  char str[100];
  int rv = p.recv(buffer,100);
  cout << "DATA Recieved on socket (" << p.getSocketValue()
      << ") Length:" << rv << endl;
  output(buffer,rv);
  if (0==rv)
   { cout << "Client exited due to zero length data" << endl;
      return(-1);
   }
  if (0==strcmp("quit",buffer)) return(1);
  if (0==strcmp("next",buffer)) return(-1);
  rv = p.send(buffer,rv);
  cout << "Try again, but in format: int string int char " << endl;
  p >> i >> str >> j >> c;
  cout << "vars:" << i << str << j << c << endl;
  cout << "done" << endl;
  p << "More Data" << i << j << c << str << endl;
  return(0);
 }
```

ПРИМЕР СЕРВЕРА (С ПОЯСНЕНИЯМИ)

```
#include :csocket.h"
```

Требуется для того, чтобы можно было использовать класс **CSocket**.

```
#include <string.h>
#include <stdlib.h>
```

Требуется для стандартных операций ввода/вывода.

```
void output(char *, int);
int service(CSocket &);

main(int argc, char *argv[])
 {
  CSocket::flagdata=0x10;
```

Используется для установки начальных значений для регистрации (Logging) при построении гнезда CSocket.

```
CSocket s;
CSocket c;
```

Созданы два гнезда — клиент и сервер.

```
int port=100;
if (2=argc) port = atoi(argv[1]);
```

Порт по умолчанию — 100, однако он может быть переопределен аргументом командной строки.

```
  s.bind(port,"127.0.0.1");
  s.listen(3);

  int quit=0;
  while (quit<1)
  {
  c = s.accept();
  while(0==quit)
  {
   quit = service(c);
  }
 }
}
```

Прием клиента и продолжение его обслуживания, пока он не возвратит 1, означающую выход для сервера, или -1, означающую переход к новому клиенту.

```
int service(CSocket & p)
 {
  char buffer[100];
  int i=-1,j=-1;
  char c='A';
  char str[100];

  int rv = p.recv(buffer,100);
```

Прием (recv) старым способом.

```
  cout << "DATA Recieved on socket (" << p.getSocketValue() << ") Length:" << rv << endl;
  output(buffer,rv);
  if (0==rv)
   { cout << "Client exited due to zero length data" << endl;
       return(-1);
         }
  if (0==strcmp("quit",buffer)) return(1);
  if (0==strcmp("next",buffer)) return(-1);

  rv = p.send(buffer,rv);
```

Отсылка старым способом.

```
  cout << "Try again, but in format: int string int char " << endl;
  p >> i >> str >> j >> c;
```

Прием (recv) с помощью потоков.

```
  cout << "vars:" << i << str << j << c << endl;
  cout << "done" << endl;

  p << "More Data" << i << j << c << str << endl;
```

Отсылка с помощью потоков.

```
return(0);
}
```

Отметим, что эта программа сервера предназначена для использования с эхо-клиентом (ECHO client).

Глава 7

Использование именованных программных каналов для сервера режима "эхо"

ВВЕДЕНИЕ

Предназначение

Главной целью данного примера является показ того, как можно создать аналогичный сервер ECHO, используя Windows NT и именованные программные каналы (Named Pipes). WIN32 API обеспечивает множество специальных функций, которые облегчают задействование модели "Клиент-Сервер" с использованием именованных программных каналов.

Клиент

Программа клиента, приведенная в данном разделе, принимает один необходимый аргумент (параметр) и позволяет иметь второй:

- *pipe-name* — данный аргумент представляет собой имя программного канала, предназначенного для подключения к нему. Формат для данного параметра позволяет вам поддерживать имя сервера и имя программного канала:

  server\pipe**name**

  Если программный канал находится на компьютере того же типа, что и клиент, то вы можете использовать:

  \\.\pipe**name**

- *Wait-Flag* — данный необязательный (optional) аргумент используется для установки исходного значения для глобальной переменной *WaitFlag*. Он указывает на то, требует ли клиент от вас нажатия <Return> до подключения к серверу.

Клиент выполняет следующие шаги:

1. Инициализирует программу.

2. Входит в цикл подключения:
   a. Входит в цикл прослушивания:
      (1) вызывает *WaitNamedPipe* для определения, находится ли сервер в режиме прослушивания программного канала
      (2) Выходит из цикла прослушивания, если канал доступен
   b. Пытается подключиться к каналу, используя *CreateFile*
   c. Выходит из цикла подключения, если подключение было успешным
3. Входит в цикл данных:
   a. Запрашивает данные для передачи на Сервер.
   b. Передает данные, используя *WriteFile*.
   c. Выходит из цикла данных, если:
      (1) программный канал испорчен;
      (2) пользователь набирает на клавиатуре "exit" ("выход").

Сервер

Программе сервера требуется один аргумент, и допускаются несколько других:

- *pipe-name* — это имя канала. Запрещается вводить в имя какие-либо специальные символы; логическая схема сервера вставляет имя таким образом, что действующий программный канал имеет правильный синтаксис.
- *MessageMode* — Определяет порядок считывания данных из программного канала. Байтовый режим означает, что вся доступная информация возвращена. Режим сообщения означает, что возвращено только следующее сообщение.
- *PipeMode* — Определяет порядок хранения данных в программном канале. Если вы храните данные в виде байтов, то вы не сможете прочитать их в режиме Message (Сообщение). Если вы храните данные в виде сообщений, то вам удастся прочитать их в виде байтов или в виде сообщений.
- *SyncMode* — Определяет состояние сервера как синхронное или асинхронное.
- *WaitFlag* — Определяет, требует ли сервер от вас нажатия <Return> до приема клиента или считывания с него данных.

Сервер выполняет следующие шаги:

1. Инициализирует программу.
2. Создает экземпляр программного канала.

3. Входит в цикл Сервера:
   a. Вызывает *ConnectNamedPipe* для ожидания подключения клиента.
   b. Обрабатывает Клиента:
      (1) Входит в цикл обработки Клиента:
         (a) Считывает данные с клиента, используя *ReadFile.*
         (b) Выходит из цикла обработки Клиента, если:
            [1] Канал испорчен,
            [2] Сервер получает команду "exit",
            [3] Сервер получает команду "bye".
         (c) Выходит из режима Сервера, если:
            [1] Сервер получает команду "quit"
         (d) Распечатывает данные, полученные с использованием функции *output.*

ИМЕНОВАННЫЕ ПРОГРАММНЫЕ КАНАЛЫ КЛИЕНТА РЕЖИМА "ЭХО" (С ПОЯСНЕНИЯМИ)

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h> // for atoi

void ErrorStop();
```

Включения

Windows.h является стандартным файлом включения, который определяет все APIs (интерфейсы прикладных программ) WIN32 на платформе Windows NT.

stdio.h и *stdlib.h* — стандарты библиотек С (язык Си) для осуществления ввода-вывода и других функций.

ErrorStop описан в дальнейшем таким образом, что может быть вызван в *main().*

```
main(int argc, char *argv[])
 {
  HANDLE hPipe;
  int rv;
  char buffer[80];
  int WaitFlag;
  int Listening=0;
```

```
int Connected=0;
int UserExit=0;
int count;
```

Локальные (местные) переменные

Локальные переменные определены следующим образом:

hPipe — хранит описатель (handle) к программному каналу;
rv — временное возвращаемое значение;
buffer — буфер объемом 80 символов, используемый для ввода
WaitFlag — обычно определяет, должен ли клиент находиться в режиме ожидания перед подключением;
Listening — используется для выхода из цикла прослушивания;
Connected — используется для выхода из цикла подключения;
UserExit — используется для выхода клиента;
count — осуществляет прием определенного числа символов, переданных командой *WriteFile*.

```
if ((2>argc)||(3<argc))
   {
    printf("Usage %s pipe-name [wait-flag:defaults to 0]\n",argv[0]);
    printf(" pipe-name \\\\server\\pipe\\name\n");
    printf(" server can be . to represent this\n");
    return(0);
   }
 if (3==argc) WaitFlag = atoi(argv[2]);
 else       WaitFlag = 0;
```

Командная строка, WaitFlag

Данная часть кода программы проверяет аргументы (параметры) командной строки и считывает переменную *WaitFlag*, если таковая присутствует. Единственным необходимым параметром является имя программного канала, к которому вы подключаетесь.

Под Windows NT программные каналы запоминаются в области глобальных имен (Global Name Space) для ЭВМ, с которой работает сервер, в специальном субкаталоге программных каналов. Имя программного канала может содержать псевдокаталоги, которые не появляются внутри диспетчера файлов (File Manager), а оканчиваются, в конечном итоге, именем программного канала. Аргумент канального имени, передаваемый клиенту, должен точно определять имя программного канала, включая имя сервера, и все псевдокаталоги. Сервер может быть определен как ".". В этом случае используется местная ЭВМ.

Например, в нижеприведенном примере Сервер — CNTRLZ, необходим каталог программных каналов, а имя программного канала — hello:

\\CNTRZ\pipe\hello

В следующем примере Сервер — OWLP2, необходим каталог программных каналов, псевдокаталог MyApp задается, а имя каталога — hello:

\\OWLP2\pipe\MyApp\port1

Сервер — локальная ЭВМ; необходим каталог программных каналов; псевдокаталог USA\Georgia\Atlanta\Cherokee задается, а имя программного канала в данном примере — Tags:

\\.\pipe\USA\Georgia\Atlanta\Cherokee\Tags

```
while(0==Connected)
 {Listening=0;
  while(0==Listening)
  {if (1==WaitFlag)
   { printf("Press Return to call WaitNamedPipe\n");
     gets(buffer);
   }
  printf("Calling WaitNamedPipe to see if a pipe is available\n");
  rv = WaitNamedPipe(argv[1],NMPWAIT_USE_DEFAULT_WAIT);
  if (0==rv)
   {switch(GetLastError())
    {case(ERROR_PIPE_BUSY): printf("The Pipe still Busy\n");
                  break;
    case(ERROR_SEM_TIMEOUT): printf("Timeout...TryAgain\n");
                              break;
    case(ERROR_FILE_NOT_FOUND):
                              printf("The Pipe does not exist\n");
                              printf("Sleeping for a second\n");
                              sleep(1000);
                              printf("Awake!\n");
                              break;
    default:                 ErrorStop();
    }
   }
  else Listening = 1;
 }
```

Состояния программного канала

Для программного канала существуют следующие состояния:

- Сервер не был запущен или не создал программный канал, поэтому программного канала просто не существует.
- Сервер создал программный канал, но еще не находится в режиме прослушивания.

- Сервер в данный момент осуществляет обработку другого клиента.
- Сервер готов к обработке клиента.

Функция *WaitNamedPipe* может использоваться для ожидания момента образования программного канала, доступного для подключения клиента. Функция *WaitNamedPipe* возвращает значение TRUE ("истинно"), если экземпляр канала доступен. В данном случае флаг *Listening* установлен в 1, а код программы переходит в следующий раздел.

Если *WaitNamedPipe* возвращает значение FALSE ("ложно"), то программный канал недоступен. Причина этого может быть в *GetLastError*. Если программный канал существует, то *WaitNamedPipe* ожидает появления доступного программного канала до его возврата. Режим ожидания продолжается до истечения лимита времени, отведенного для него. Клиент использует время ожидания по умолчанию, определенное сервером во время создания программного канала.

В случае, когда программного канала не существует, вы ждете 1 секунду перед продолжением, т.к. *WaitNamedPipe* возвращается немедленно, если программный канал не существует.

```
printf("A Pipe instance is available\n");
if (1==WaitFlag)
  { printf("Press Return to call CreateFile\n");
    gets(buffer);
  }
printf("Calling CreateFile to connect to the pipe\n");
hPipe = CreateFile(argv[1],GENERIC_WRITE,0,NULL, OPEN_EXISTING,0,NULL);
if (hPipe == INVALID_HANDLE_VALUE)
   {switch(GetLastError())
     {case(ERROR_PIPE_BUSY):      printf("Somebody stole my pipe!\n");
                          break;
      default:              ErrorStop();
   }
  }
 else Connected=1;
```

Цикл прослушивания (Listen Loop)

Возвращение TRUE ("истинно") с помощью *WaitNamedPipe* еще не означает, что ваш клиент может получить программный канал. Возможен вариант, когда другой клиент только что стартовал. В этом случае клиент видит, что программный канал доступен и забирает его до того, как ваш клиент получит такую же возможность. Вот почему существует цикл прослушивания (Listening Loop), встроенный в цикл подключения (Connected Loop).

Когда вы вышли из цикла прослушивания, постарайтесь занять программный канал, используя *CreateFile*. Если другой клиент успел занять программный канал раньше вас, вы получите сообщение об ошибке и возврат к началу в цикле ожидания до образования другого экземпляра доступного программного канала. Если вам удалось подключиться к программному каналу, то описатель (handle), возвращенный с помощью *CreateFile*, становится доступным. Таким образом, вы задаете переменную *Connected* для выхода из цикла подключения. После этого вы можете продолжить выполнение программ.

```
printf("Enter data to send to the server, exit to quit\n");
 while(0==UserExit)
  {
   printf("Client>");
   gets(buffer);
   rv = WriteFile(hPipe, buffer, strlen(buffer)+1, &count, NULL);
   if (0==rv)
    {switch(GetLastError())
     {    case(ERROR_PIPE_NOT_CONNECTED):
                          printf("The pipe is no longer connected\n");
                          printf("Client exiting...\n");
                          return(0);
      default:          ErrorStop();
     }
    }
   else
     {printf("[%d] characters transmitted\n",count);
      if (0==strcmp("exit",buffer)) UserExit = 1;
    }
  }
 printf("Pipe being closed\n");
 CloseHandle(hPipe);
 if (1==WaitFlag)
   { printf("Press Return to Exit Client\n");
     gets(buffer);
   }
 }
```

Отправка данных: *WriteFile*

Теперь, после того как вы подключились к программному каналу, вы можете пригласить пользователя ввести данные, а затем отправить их на сервер, используя функцию *WriteFile*. Если *WriteFile* возвращает FALSE ("ложно"), то вы проверяете *GetLastError()* для выяснения типа имевшей место ошибки.

Если данные были успешно переданы, вы просматриваете их и определяете, передана или нет команда "exit" ("выход"). Если да, то клиент выходит, установив флаг *UserExit* в 1. Эта операция вызывает отказ цикла *while*, и клиент выходит из цикла.

```
void ErrorStop()
  {
   DWORD rv = GetLastError();
   char *Pointer;

   printf("Error:%d\n", rv);
   FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_IGNORE_INSERTS |
            FORMAT_MESSAGE_FROM_SYSTEM,
              0,(DWORD)rv,0
             ,(char *) &Pointer,0,NULL);
 printf("%s\n",Pointer);
 LocalFree(Pointer);
 ExitThread(0);
}
```

ErrorStop()

Данная функция используется для осуществления запроса Windows NT о последней ошибке и обеспечения распечатки сообщения об ошибке. Функция *FormatMessage* получает команды: на выделение памяти для сообщения об ошибке, на просмотр таблицы сообщений об ошибках системы и на пропуск любых форматирующих символов в сообщении об ошибке. Передается указатель на переменную, измененную с помощью *FormatMessageR*, и *FormatMessage* размещает адрес буфера, который был выделен функцией, в эту переменную. В данном случае адрес *Pointer* передается в *FormatMessage*.

После возвращения *FormatMessage* вы распечатываете сообщение, а затем освобождаете память, выделенную с помощью *FormatMessage*. После этого функция *ErrorStop* прерывает цепочку, которая содержала ошибку.

СЕРВЕР NPECHO (С ПОЯСНЕНИЯМИ)

```
 #include <windows.h>
 #include <stdio.h>
 #include <stdlib.h>

 void ErrorStop();
 int ProcessClient();
 void output(char*);
HANDLE hPipe;

  int SyncMode;
  int PipeMessage;
  int MessageMode;
  int WaitFlag;
```

Включения, переменные

Сервер имеет заголовочные файлы, аналогичные файлам клиента, а *ErrorStop* аналогичную клиенту функцию. Однако сервер имеет две другие функции, которые определены напрямую: *ProcessClient* и *output*.

Переменная *hPipe* используется для хранения дескриптора, или описателя (handle), программного канала, который создан для связи с клиентами.

Глобальные переменные *SyncMode*, *PipeMode* и *MessageMode* используются для запуска программного канала сервера. Переменная *WaitFlag* используется для указания на то, требует ли сервер от пользователя нажатия <Return> (<Возврат>) до получения данных от пользователя.

```
int main(int argc, char *argv[])
  {int ServerStopFlag =0;
  char buffer[80];
  char input[80];

  if ((2>argc)||(6<argc))
   {
   printf("Usage: %s pipe-name [message-mode][pipe-mode][sync- mode] [wait-mode]\n",argv[0]);
   printf("   messageMode: How the data is read out of the pipe  [0 default]\n");
   printf("   pipeMessage: How data is stored in pipe [0 default]\n");
   printf("   syncMode:  does the pipe block? [1 default]\n");
   printf("   waitFlag:  wait before actions? [0 default]\n");
   printf("\n");
   return(0);
   }
  if (3<=argc) MessageMode = atoi(argv[2]);
  else        MessageMode = 0;
  MessageMode = (MessageMode==0)? PIPE_READMODE_BYTE : PIPE_READMODE_MESSAGE;

  if (4<=argc) PipeMessage = atoi(argv[3]);
  else        PipeMessage = 0;
  PipeMessage = (PipeMessage==0)? PIPE_TYPE_BYTE : PIPE_TYPE_MESSAGE;

  if (5<=argc) SyncMode = atoi(argv[4]);
  else        SyncMode = 1;
  SyncMode  = (SyncMode==0)? PIPE_NOWAIT : PIPE_WAIT;

  if (6<=argc) WaitFlag = atoi(argv[5]);
  else        WaitFlag = 0;
```

Инициализация, обработка командной строки

Данная часть кода программы используется для инициализации программы и считывания аргументов командной строки. Переменные

MessageMode, *PipeMessage* и *SyncMode* считываются как двоичное число — 1 или 0, — а затем изменяются на правильно определенные значения (*#defined*), как это требуется функцией *CreateNamedPipe*. Если значение *MessageMode* равно 0, то эта переменная изменяется на PIPE_READMODE_BYTE. В противном случае она изменяется на PIPE_READMODE_MESSAGE.

Когда программный канал создается как программный канал с режимом MESSAGE, он может считываться как в режиме сообщений, так и в байтовом режиме. Если программный канал создан для работы в байтовом режиме (BYTE mode), он не может считываться в режиме сообщений, потому что в этом режиме программный канал не сможет указать, где одно сообщение заканчивается, а другое начинается.

При считывании программного канала, работающего в режиме сообщений, вы получаете только следующее сообщение в канале даже в том случае, когда в нем проходит более одного сообщения. При считывании в байтовом режиме вы получите то количество байтов информации, которые вы указали, или все байты информации, находящиеся в программном канале.

Если программный канал находится в режиме ожидания (wait mode), сервер заблокирован до тех пор, пока не завершится операция чтения, означающая, что клиент записал данные. В противном случае функция *read* не выполняется, выдавая ошибку, которая указывает на отсутствие данных.

```
sprintf(buffer,"\\\\.\\pipe\\%s",argv[1]);
 printf("Calling CreateNamedPipe('%s'...)\n",buffer);
 printf("  MessageMode:%s\n", (PIPE_READMODE_BYTE == MessageMode)?"Byte":"Message");
 printf("  PipeMode  :%s\n", (PIPE_TYPE_BYTE == PipeMessage)?"Byte":"Message");
 printf("  WaitMode  :%s\n", (PIPE_NOWAIT == SyncMode)?"NoWait":"Wait");
```

Формат именованного программного канала

Имя программного канала, переданное серверу, вводится в буфер таким образом, чтобы оно при передаче на *CreateNamedPipe* имело правильный формат. Формат для имени программного канала, переданного на *CreateNamedPipe*, имеет вид:

\\.\pipe**name**

Следует заметить, что сервер может создать именованный программный канал лишь на локальной ЭВМ. Часть **name** (имени) именованного канала, при необходимости, может включать в себя псевдокаталоги:

\\.\pipe\MyApp\port1

или

\\.\pipe\USArea\Georgia\Atlanta\Cherokee\Tags

Аргументы командной строки для вышеперечисленных примеров имеют вид:

name

MyApp\port1

USArea\Georgia\Atlanta\Cherokee\Tags

Следует заметить, что имена нечувствительны к регистру символов, когда клиент пытается открыть существующий канал.

```
hPipe = CreateNamedPipe(buffer,PIPE_ACCESS_DUPLEX,
    MessageMode|PipeMessage|SyncMode,1,0,0,10000,NULL);

if (INVALID_HANDLE_VALUE==hPipe) ErrorStop();
```

CreateNamedPipe (Создание именованного программного канала)

Функция *CreateNamedPipe()* принимает следующие аргументы:

- Имя программного канала, который должен быть создан.
- Режим открытого программного канала (Pipe Open Mode):
  - — DUPLEX — означает потоки данных в обоих направлениях;
  - — INBOUND — означает поток данных только от клиента к серверу;
  - — OUTBOUND — означает поток данных только от сервера к клиенту.
- Режим создания канала:
  - — Message, Byte — В каком виде данные хранятся в программном канале?
  - — ReadMessage, ReadByte — В каком виде данные считываются из программного канала?
  - — Wait, NoWait — Каким образом осуществляется блокирование?
- Max Instances — Какое количество экземпляров программного канала может существовать?
- Out Buffer size — Размер выходного буфера.
- In Buffer size — Размер входного буфера.
- Default Timeout — Как долго клиент ожидает, если он использует канал по умолчанию?
- Security attributes — Ввиду того что канал является объектом NT, может ли он быть защищен?

Атрибут PIPE_ACCESS_DUPLEX присваивается создаваемому программному каналу таким образом, что, при необходимости, связь можно осуществлять в обоих направлениях. Клиент может отправлять данные серверу, а сервер может отправлять информацию клиенту.

В данном примере количество экземпляров программных каналов установлено равным 1, т.к. сервер обслуживает только одного клиента за один раз. Следующий пример иллюстрирует порядок обслуживания многих клиентов под Windows NT.

Если возвращенный дескриптор (handle) неверен, вы обращаетесь к *ErrorStop* для распечатки сообщения об ошибке и выходите из программы.

```
while(0 == ServerStopFlag)
  {
   int rv;

   if (1==WaitFlag)
    {
     printf("press return to wait for a client to connect\n");
     gets(input);
    }
   printf("Calling ConnectNamedPipe to accept a new client\n");
   rv = ConnectNamedPipe(hPipe,NULL);
   if (0==rv)
     {switch(GetLastError())
      {case(ERROR_PIPE_CONNECTED):
       printf("Client connected before server called ConnectNamedPipe\n");
       rv = 1;
       break;
      case(ERROR_PIPE_LISTENING):
       printf("The Server is Listening for a Client in Non Blocking Mode,\n");
        printf("But a Client is not available, server will sleep for 10 seconds\n");
       Sleep(10000);
       break;
      default :ErrorStop();
      }
     }
          if (0!=rv)
            {printf("A Client is available\n");
             ServerStopFlag = ProcessClient();
            }
  }
 }
```

Обработка сервера

Данный раздел кода программы представляет собой основную часть обработки сервера. Переменная *ServerStopFlag* используется для указания на то, когда сервер должен осуществить выход. Если *WaitFlag* задействован, сервер приглашает пользователя нажать <Return> (<Возврат>), перед тем как продолжать прием клиента. Таким образом, вы получаете шанс запустить сервер, создать программный канал и смоделировать событие, когда клиент подключается к программному каналу до вызова сервером *ConnectNamedPipe*.

Функция *ConnectNamedPipe* осуществляет операции, аналогичные функции *accept* в примере гнезд. В случае с неблокированным программным каналом *ConnectNamedPipe* немедленно возвращается с ошибкой, если клиент недоступен. В этом случае вы ждете в течение 10 секунд, чтобы избежать распечатки тысяч сообщений на экране.

Следует заметить, что функция *ConnectNamedPipe* не выполняется, если клиент подключается к каналу до обращения сервера к *ConnectNamedPipe*. В данном случае вы распечатываете сообщение и вручную изменяете возвращаемое значение на TRUE ("истинно"), так что вы можете обрабатывать клиента.

Единственным вариантом, когда происходит обработка клиента, является тот случай, когда возвращаемое значение переменной *rv* установлено в TRUE ("истинно").

```
int ProcessClient()
 {
  int Length,rv;
  char buffer[100];
  while(1==1)
   { if (1==WaitFlag)
     {
      printf("press return to read data from the client\n");
      gets(buffer);
   }
   rv = ReadFile(hPipe,buffer,sizeof(buffer),&Length,NULL);
```

ProcessClient()

Данная функция используется для обработки клиента. Если сервер обнаружил подключение клиента, он вызывает эту функцию для обработки клиента. Для продолжения считывания строк от клиента задействуется бесконечный цикл до тех пор, пока не происходит событие, вызывающее возврат функции.

Функция *ReadFile* используется для считывания данных из программного канала. Функция *ReadFile* требует следующих параметров:

- дескриптор (handle) файла, который считывается, т.е. *hPipe* — глобальная переменная;
- буфер, в который происходит считывание данных;
- такой размер буфера, который *ReadFile* не переполняет в случае, когда доступными становятся большие массивы данных;
- указатель на *int*, который используется для того, чтобы содержать в себе число байтов данных, которые считываются;
- указатель на структуру с перекрытием, если вы используете ввод/вывод с перекрытием — в данном случае этого нет.

```
if (0==rv)
    {switch(GetLastError())
     {case(ERROR_BROKEN_PIPE):
        printf("The client has unexpectedly broken the pipe\n");
        DisconnectNamedPipe(hPipe);
        return(0);
      case(ERROR_NO_DATA):
        printf("There is no data available at the pipe\n");
        printf("Server sleeping for 3 seconds\n");
        Sleep(3000);
        break;
      default: ErrorStop();
        printf("Server sleeping for 10 seconds\n");
        Sleep(10000);
    }
  }
```

Обработка ошибок

Если функция *ReadFile* возвращает ошибку, вам следует проверить, не является ли ошибка следствием следующих событий:

- Испорченный программный канал (Broken pipe) — клиент закрыл программный канал на своем конце. Если это так, то отключите клиента от этого конца программного канала и возвратите его из обработки. Эта операция вызовет режим ожидания для подключения другого клиента.
- Отсутствие данных — программный канал сервера находится в неблокирующем режиме (NOWAIT). Подождите 3 секунды появления данных и сделайте еще одну попытку. Функция *break* приводит к тому, что *switch* выходит, что продолжается и в дальнейшем.
- Другая ошибка — подождите 10 секунд и сделайте еще одну попытку. Это даст вам возможность увидеть ошибку.

```
  if (0==strcmp(buffer,"quit"))
      {printf("received 'quit' command, Server Exiting\n");
       DisconnectNamedPipe(hPipe);
       return(1);
      }
    if (0==strcmp(buffer,"exit"))
      {printf("received 'exit' command, Waiting for next client\n");
       DisconnectNamedPipe(hPipe);
       return(0);
      }
    if (0==strcmp(buffer,"bye"))
      {printf("received 'bye' command, client being disconnected\n");
       DisconnectNamedPipe(hPipe);
       return(0);
      }

    printf("Server Received:");
    output(buffer);
  }
}
```

Встроенные команды

Проверьте буфер на наличие встроенных команд и выполните соответствующие функции:

- *quit* (конец сеанса) — Возвращает 1 обратно вызывающей программе, что приводит к выходу сервера (exit);
- *exit* (выход из обработки) — Отключает текущего клиента и возвращается для обработки другого. Клиент выходит из обработки;
- *bye* (закрытие) — Отключает текущего клиента и возвращается для обработки другого. Клиент не выходит из обработки. Таким образом, моделируется ситуация того, что происходит с клиентом, когда сервер прекращает обработку програм-много канала.

Если не было найдено ни одной встроенной команды, то следует распечатать данные, полученные от клиента.

```
void ErrorStop()
  {
   DWORD rv = GetLastError();
   char *Pointer;

   printf("Error:%d\n", rv);
   FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                 FORMAT_MESSAGE_IGNORE_INSERTS |
```

```
                    FORMAT_MESSAGE_FROM_SYSTEM,
                     0,(DWORD)rv,0,
                    (char *) &Pointer,0,NULL);
  printf("%s\n",Pointer);
  LocalFree(Pointer);
 }

void output(char *buffer)
 {
 while (*buffer > 0)
  {
   if (31>*buffer) printf("\\%03d",*buffer);
   else      printf("%c",*buffer);

   buffer++;
  }
 printf("\n");
 }
```

Функции помощника

Функции помощника используются для показа ошибок или символьных строк, содержащих встроенные управляющие символы.

ПОСТРОЕНИЕ ПРОГРАММ В SDK

Makefile (Формирующий файл)

Используйте следующий формирующий файл для построения клиента и сервера, если вы пользуетесь SDK и не используете язык Visual C++.

```
!include <ntwin32.mak>

 cflags=$(cflags) -DDEVL=1 -D_MT=1 -Gz

 all: server.exe client.exe

 server.exe: server.obj
   $(link) $(linkdebug) $(conflags) -out:$*.exe $** $(conlibs)

 client.exe: client.obj
   $(link) $(linkdebug) $(conflags) -out:$*.exe $** $(conlibs)

 client.obj: client.c
   $(cc) $(cflags) $(cvars) $(cdebug) client.c

 server.obj: server.c
   $(cc) $(cflags) $(cvars) $(cdebug) server.c
```

Построение файла регистрации (Log File)

Приведенное ниже является файлом регистрации построения готовых к выполнению программ с использованием SDK:

```
Microsoft (R) Program Maintenance Utility  Version 1.40 Copyright (c) Microsoft
Corp 1988-93. All rights reserved.

   cl386 -c -W3 -D_X86_=1 -DDEVL=1 -D_MT=1 -Gz -DWIN32 -D_MT -Zi -Od server.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 8.00
Copyright (c) Microsoft Corp 1984-1993. All rights reserved.

server.c
server.c(17) : warning C4007: 'main' : must be 'cdecl'
f:\msvcnt\bin\CL386.EXE: warning: invoking f:\msvcnt\bin\CL.EXE
    link32 -debug:full -debugtype:cv -align:0x1000 -subsystem:console -
entry:mainCRTStartup -out:server.exe server.obj libc.lib kernel32.lib
Microsoft (R) 32-Bit Executable Linker Version 1.00
Copyright (C) Microsoft Corp 1992-93. All rights reserved.

f:\msvcnt\bin\LINK32.EXE: warning: invoking f:\msvcnt\bin\LINK.EXE
     cl386 -c -W3 -D_X86_=1 -DDEVL=1 -D_MT=1 -Gz -DWIN32 -D_MT -Zi -Od client.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 8.00
Copyright (c) Microsoft Corp 1984-1993. All rights reserved.

client.c
client.c(8) : warning C4007: 'main' : must be 'cdecl'
f:\msvcnt\bin\CL386.EXE: warning: invoking f:\msvcnt\bin\CL.EXE
     link32 -debug:full -debugtype:cv -align:0x1000 -subsystem:console -
entry:mainCRTStartup -out:client.exe client.obj libc.lib
kernel32.lib
Microsoft (R) 32-Bit Executable Linker Version 1.00
Copyright (C) Microsoft Corp 1992-93. All rights reserved.

f:\msvcnt\bin\LINK32.EXE: warning: invoking f:\msvcnt\bin\LINK.EXE
```

ИСПОЛЬЗОВАНИЕ ЯЗЫКА VISUAL C++ ДЛЯ ПОСТРОЕНИЯ ФАЙЛОВ

Создание нового файла проекта (Project File) и построение готовых к выполнению программ

Для создания нового проекта на языке Visual C++, используйте шаги, приведенные в примере ECHO. Приведенный ниже список характеризует эти шаги:

- Для создания нового проекта используйте выборку меню Project New.
- Идентифицируйте проект как Console Application. Имейте в виду, что данный пример — не для Windows 3.1.
- Выберите файл Server.c, который должен быть добавлен к проекту.

- Установите опции проекта, используя выборку меню Options Project.
- В Linker Options не требуется вносить никаких изменений.
- Постройте готовую к выполнению программу, щелкнув мышью по клавише *Build.*
- Установите аргументы командной строки, используя пункт из Options Debug: **port100**
- Выполните вышеперечисленные шаги для добавления Client.c к другому проекту и постройте готовую к выполнению программу.
- Установите параметры командной строки в \\.**pipe\port100**

Пример программы

В приведенной ниже программе запускается сервер, а затем в разные моменты времени стартуют три клиента. Файлы регистрации иллюстрируют последовательность событий.

ПРИМЕРЫ ФАЙЛОВ РЕГИСТРАЦИИ

Файл регистрации сервера

```
Calling CreateNamedPipe('\\.\pipe\port1'...)
  MessageMode:Byte
  PipeMode  :Byte
  WaitMode  :Wait
Calling ConnectNamedPipe to accept a new client
A Client is available
Server Received:Hello From One
Server Received:Hello Again from One
Server Received:Goodbye from One
received 'bye' command, client being disconnected
Calling ConnectNamedPipe to accept a new client
A Client is available
Server Received:Hello on Two
Server Received:HA! I beat Three to the Pipe
Server Received:Goodbye(Exit)
received 'exit' command, Waiting for next client
Calling ConnectNamedPipe to accept a new client
A Client is available
Server Received:Hello on Three
Server Received:Goodbye (quit)
received 'quit' command, Server Exiting
```

Сервер запускается, Клиент 1 подключается, Клиенты 2 и 3 ожидают. После отключения Клиента 1 оба Клиента 2 и 3 видят именованный программный канал; Клиент 2 подключается первым.

После окончания работы Клиента 2 подключается Клиент 3. Клиент 3 вызывает выход сервера.

Файл регистрации Клиента 1

```
Calling WaitNamedPipe to see if a pipe is available
A Pipe instance is available
Calling CreateFile to connect to the pipe
Enter data to send to the server, exit to quit
Client>Hello From One
[15] characters transmitted
Client>Hello Again from One
[21] characters transmitted
Client>Goodbye from One
[17] characters transmitted
Client>bye
[4] characters transmitted
Client>More data???
The pipe is no longer connected
Client exiting...
```

Файл регистрации Клиента 2

```
WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
A Pipe instance is available
Calling CreateFile to connect to the pipe
Enter data to send to the server, exit to quit
Client>Hello on Two
[13] characters transmitted
Client>HA! I beat Three to the Pipe
[29] characters transmitted
Client>Goodbye(exit)
[14] characters transmitted
Client>exit
[5] characters transmitted
Pipe being closed
```

Файл регистрации Клиента 3

```
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
```

```
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
A Pipe instance is available
Calling CreateFile to connect to the pipe
Somebody stole my pipe!
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
A Pipe instance is available
Calling CreateFile to connect to the pipe
Enter data to send to the server, exit to quit
Client>Hello on Three
[15] characters transmitted
Client>Goodbye (quit)
[15] characters transmitted
Client>quit
[5] characters transmitted
Client>More Data???
The pipe is no longer connected
Client exiting...
```

Глава 8

Программа многоклиентских именованных программных каналов

ВВЕДЕНИЕ

Назначение

Главная цель данного примера программы — показать, как надо пользоваться сервером, способным обслуживать большое количество клиентов под Windows NT WIN32 API (интерфейсы прикладных программ) с использованием именованных программных каналов.

Клиент

Программа клиента для данного примера абсолютно идентична предыдущей, приведенной в примере NPECHO, с одним лишь исключением. Данная программа имеет дополнительный оператор *printf*, который указывает пользователю на то, что первая строка данных, передаваемая серверу, используется последним как текст идентификатора модуля клиента (Client ID). При необходимости предыдущий модуль клиента может быть использован без видоизменений.

Код программы, добавленный к новому модулю клиента

Данный код программы был преобразован в модуль клиента NPMULTI, что делает его несколько другим.

Исходный код программы:

```
.
.
.
 else Connected=1;
}
  printf("Enter data to send to the server, exit to quit\n");

while(0==UserExit)
  { printf("Client>");
.
.
```

Новый код программы:

```
•
•
•
 else Connected=1;
}
  printf("The first line of data you send will become your Client id\n");
  printf("Enter data to send to the server, exit to quit\n");

while(0==UserExit)
  { printf("Client>");
•
•
•
```

Сервер

Сервер использует большую часть того же самого кода программы, что и в предыдущем примере. Основные изменения приведены ниже:

Функция *CreateNamedPipe* перенесена в *основной* (*main*) цикл Сервера. Каждый раз, когда модуль клиента подключается к серверу, функция *ProcessClient* вызывается как отдельная цепочка (thread). Основная цепочка образует канал обратной связи, который создает другой экземпляр программного канала, для того чтобы дать возможность подключиться новому клиенту.

ИЗМЕНЕНИЯ В КОДЕ ПРОГРАММЫ СЕРВЕРА

Приведенное ниже характеризует изменения, сделанные в сервере, которые делают его цепочечно-значимым (thread-aware).

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

void ErrorStop();
DWORD ProcessClient(HANDLE);
void output(char*);
int ServerStopFlag =0;
```

Переменная *ServerStopFlag* преобразуется в глобальную переменную.

В примере ECHO значение возврата *ProcessClient* используется для установки значения *ServerStopFlag*. В примере MULTI *ProcessClient* вызывается как цепочка (thread), поэтому она не может вернуть значение вызывающей программе. Вместо этого, если новая цепочка примет

решение, что главный сервер должен выйти из цикла, она выполняет функцию *ExitProcess* для завершения всего процесса сервера.

```
  int SyncMode;
  int PipeMessage;
  int MessageMode;
  int WaitFlag;
  int maxClients;
int main(int argc, char *argv[])
 {

  HANDLE hPipe;
  char buffer[80];
  char input[80];
```

Теперь появляется новый аргумент (параметр): *maxClients.*

Данный аргумент используется для определения максимального числа клиентов, которое может обслуживаться одновременно.

```
if ((3>argc)||(7<argc))
    {
       printf("Usage: %s pipe-name [maxClients][message-mode] [pipe-mode] [sync-mode]
[wait-mode]\n",argv[0]);
     printf("  maxClients: The maximum clients that can be connected [5 default]\n");
     printf("  messageMode: How the data is read out of the pipe [0 default]\n");
     printf(" pipeMessage: How data is stored in pipe [0 default]\n");
     printf(" syncMode:  does the pipe block? [1 default]\n");
     printf(" waitFlag:  wait before actions? [0 default]\n");
     printf("\n");
     return(0);
    }

if (3<=argc) maxClients = atoi(argv[2]);
else       maxClients = 5;

if (4<=argc) MessageMode = atoi(argv[3]);
else       MessageMode = 0;

MessageMode = (MessageMode==0)? PIPE_READMODE_BYTE : PIPE_READMODE_ MESSAGE;

if (5<=argc) PipeMessage = atoi(argv[4]);
else       PipeMessage = 0;

PipeMessage = (PipeMessage==0)? PIPE_TYPE_BYTE : PIPE_TYPE_MESSAGE;

if (6<=argc) SyncMode = atoi(argv[5]);
else       SyncMode = 1;

SyncMode  = (SyncMode==0)? PIPE_NOWAIT : PIPE_WAIT;
```

```
if (7==argc) WaitFlag = atoi(argv[6]);
else        WaitFlag = 0;

sprintf(buffer,"\\\\.\\pipe\\%s",argv[1]);
printf("Calling CreateNamedPipe('%s'...)\n",buffer);
printf(" MessageMode:%s\n",
  (PIPE_READMODE_BYTE == MessageMode)?"Byte":"Message");
printf(" PipeMode :%s\n",
  (PIPE_TYPE_BYTE == PipeMessage)?"Byte":"Message");
printf(" WaitMode  :%s\n",
  (PIPE_NOWAIT == SyncMode)?"NoWait":"Wait");

while(0 == ServerStopFlag)
  {
```

Функция *CreateNamedPipe* перемещается внутри цикла *while*. В примере ECHO был реализован только один случай образования программного канала. В данном примере для каждого клиента, который подключается, необходимо создавать новый экземпляр программного канала. Это осуществляется путем помещения вызова функции *CreateNamedPipe* внутрь цикла *while*.

```
DWORD ID;
HANDLE hThread;
```

Таким образом, определены две новые локальные переменные (Local variables). При создании новой цепочки вы получите описатель (handle) новой цепочки, также как и связанный с цепочкой идентификатор (ID). Даже если вам эти переменные не нужны, вы описали переменные для хранения информации.

```
int rv;
  printf("Calling CreateNamedPipe to create a new pipe instance\n");
```

Обращение к *CreateNamedPipe* включает в себя новый аргумент *maxClient*.

Он передается как максимальный номер экземпляра, показывающий максимальное число экземпляров программных каналов, которое может быть создано.

```
hPipe =  CreateNamedPipe(buffer,PIPE_ACCESS_DUPLEX,
        MessageMode|PipeMessage|SyncMode,
       maxClients,0,0,10000,NULL);

 if (INVALID_HANDLE_VALUE==hPipe)
```

В условиях существования максимального номера экземпляра существует метод создания программного канала.

Так как вы создаете не один, а несколько экземпляров программного канала, вы знаете момент создания большего числа каналов, чем вам разрешено. Если вы оказались в данной ситуации, вы ждете 10 секунд и делаете еще одну попытку. В конечном счете, одна из цепочек отключает своего клиента и, таким образом, число открытых экземпляров сокращается, так что вы можете создать еще один без опасности совершить ошибку.

```
{switch(GetLastError())
    {case(ERROR_PIPE_BUSY):
          printf("The Maximum number of clients have been reached\n");
          printf("Main Server sleeping for 10 Seconds\n");
          Sleep(10000);
    break;
default :ErrorStop();
                    return(0);
}
```

Если во время действия *CreateNamedPipe* возникает ошибка, вы продолжаете идти к вершине цикла.

Без этого оператора продолжения работы вы пытались бы выполнить *ConnectNamedPipe*, используя недействительный описатель, который был возвращен из *CreateNamedPipe*. Это происходит в том случае, когда вы достигли максимального количества экземпляров программного канала.

```
  continue; // continue to the top of the while loop
}
```

Для распечатки экземпляра программного канала, который вы только что создали, добавляется новый оператор *printf*.

```
else printf("Created Pipe instance %d\n",hPipe);
  if (1==WaitFlag)
   {
    printf("press return to wait for a client to connect\n");
    gets(input);
   }
  printf("Calling ConnectNamedPipe on pipe %d to accept a new client\n",hPipe);
  rv = ConnectNamedPipe(hPipe,NULL);
  if (0==rv)
   {switch(GetLastError())
    {case(ERROR_PIPE_CONNECTED):
      printf("Client connected before server called ConnectNamedPipe\n");
```

```
    rv = 1;
    break;
   case(ERROR_PIPE_LISTENING):
    printf("The Server is Listening for a Client in Non Blocking Mode,\n");
    printf("But a Client is not available, server will sleep for 10 seconds\n");
    Sleep(10000);
    break;

   default :ErrorStop();
  }
 }
     if (0!=rv)
       {printf("A Client is available\n");
```

Функция *CreateThread* используется для вызова подпрограммы *ProcessClient.*

Функция *CreateThread* принимает следующие аргументы:

• Атрибуты безопасности	NULL = без защиты
• Размер стека	0 = использование по умолчанию
• Начальный адрес	Начало с ProcessClient
• Аргумент	Ввести описатель канала как аргумент
• Указатель переменной DWORD	Создать цепочку (CreateThread) в соответствии с имеющимся ThreadID

```
hThread = CreateThread(NULL,0,
            (LPTHREAD_START_ROUTINE) ProcessClient,
            hPipe,0,&ID);

 if (NULL == hThread)
   {switch(GetLastError())
       {default :ErrorStop();
       }
   }

       }
  }
}
DWORD ProcessClient(HANDLE hPipe)
```

Теперь функция *ProcessClient* принимает аргумент.

Функция, указанная как адрес для *CreateThread* принимает единственный аргумент DWORD. Как правило, это указатель структуры, содержащей все аргументы функции, но в данном случае единственные данные, которые вам потребуются, — это описатель канала. Он вписывается в DWORD. Таким образом, это единственный аргумент.

```
{
 int Length,rv;
 char buffer[100];
 char idClient[100];
```

В контексте с новой цепочкой определяется новая локальная переменная.

Ввиду того что цепочка имеет свой собственный стек, локальная переменная является уникальной и различной в каждой цепочке. Только глобальные и статические переменные совместно используются цепочками. Описывается локальная переменная *idClient*. Данная переменная содержит текст первого сообщения, полученного этой цепочкой. Этот текст используется для идентификации клиента в будущих операторах *printf*. Это позволяет обрабатывать на сервере в различных цепочках большое количество сообщений, поступающих от большого числа клиентов. В то же самое время это обеспечивает осуществление логической операции регистрации, так как по каждому сообщению можно определить, какому клиенту оно принадлежит.

```
rv = ReadFile(hPipe,buffer,sizeof(buffer),&Length,NULL);
if (0==rv)
    {switch(GetLastError())
     {case(ERROR_BROKEN_PIPE):
         printf("The client[%d] has unexpectedly broken the pipe\n",hPipe);
         DisconnectNamedPipe(hPipe);
                  CloseHandle(hPipe);
         return(0);
     default: ErrorStop();
     }
    }
  else strncpy(idClient,buffer,100);

  printf("Client '%s' has connected to pipe %d\n",idClient,hPipe);
```

Один из сегментов (Section) кода программы используется для считывания и запоминания первого сообщения, поступившего от клиента.

```
while(1==1)
   {
    if (1==WaitFlag)
     {
      printf("press return to read data from the client\n");
      gets(buffer);
     }
    rv = ReadFile(hPipe,buffer,sizeof(buffer),&Length,NULL);
    if (0==rv)
     {switch(GetLastError())
```

```
{case(ERROR_BROKEN_PIPE):
    printf("The client[%s] has unexpectedly broken the pipe\n",idClient);
    DisconnectNamedPipe(hPipe);
        CloseHandle(hPipe);
```

Данная часть программы отключает клиента и закрывает модуль описателя (handle) программного канала.

Когда клиент финишировал, необходимо закрыть модуль описателя программного канала. После осуществления выхода из THREAD описатель не исчезает, так как он является ресурсом PROCESS. Если цепочка не закрывает программный канал, то она остается до тех пор, пока не прекратится весь обслуживающий процесс. Нефункциональный программный канал осуществляет счет в качестве одного из экземпляров программного канала, считающего до значения *maxClients*. Так как клиенты продолжают подключаться и отключаться, вы в конечном счете выходите из программных каналов для обслуживания клиентов. В этом случае вы получите количество *maxInstances* зависших нефункциональных программных каналов. Поэтому вы не можете использовать *CreateNamedPipe* для создания еще хотя бы одного программного канала.

```
    return(0);
case(ERROR_NO_DATA):
    printf("There is no data available at the pipe\n");
    printf("This Thread sleeping for 3 seconds\n");
    Sleep(3000);
    break;
default: ErrorStop();
    printf("Thread exiting....\n");
      CloseHandle(hPipe);
    return(0);
}
}

if (0==strcmp(buffer,"quit"))
  {
  // exit the server
  printf("received 'quit' command on client [%s], Server Exiting\n",idClient);
  DisconnectNamedPipe(hPipe);
   ExitProcess(1);
  return(1);
  }
if (0==strcmp(buffer,"exit"))
  {
  // exit this client
```

```
  printf("received 'exit' command on client  [%s]\n",idClient);
  CloseHandle(hPipe);
  DisconnectNamedPipe(hPipe);
  return(0);
  }
if (0==strcmp(buffer,"bye"))
  {
  // exit this client
   printf("received 'bye' command on client [%s], client being disconnected\n",idClient);
  CloseHandle(hPipe);

  DisconnectNamedPipe(hPipe);
   return(0);
  }

  printf("%s:",idClient);
  output(buffer);
  }
}

void ErrorStop()
 {
  DWORD rv = GetLastError();
  char *Pointer;

  printf("Error:%d\n", rv);
  FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
      FORMAT_MESSAGE_IGNORE_INSERTS |
      FORMAT_MESSAGE_FROM_SYSTEM,
       0,(DWORD)rv,0,
       (char *) &Pointer,0,NULL);
  printf("%s\n",Pointer);
  LocalFree(Pointer);
 }

void output(char *buffer)
 {
 while (*buffer > 0)
   {
    if (31>*buffer) printf("\\%03d",*buffer);
    else   printf("%c",*buffer);

    buffer++;
   }
 printf("\n");
 }
```

ПОСТРОЕНИЕ ГОТОВЫХ К ВЫПОЛНЕНИЮ ПРОГРАММ С ПОМОЩЬЮ SDK

Формирующий файл

Нижеприведенный формирующий файл (makefile) аналогичен примеру NPECHO, но он является многократно цепочечно-безопасным (multiple thread-safe) благодаря расширениям **mt** некоторых макрокоманд.

```
!include <ntwin32.mak>

cflags=$(cflags) -DDEVL=1 -D_MT=1 -Gz

all: server.exe client.exe

server.exe: server.obj
  $(link) $(linkdebug) $(conflags) -out:$*.exe $** $(conlibsmt)

client.exe: client.obj
  $(link) $(linkdebug) $(conflags) -out:$*.exe $** $(conlibsmt)

client.obj: client.c
  $(cc) $(cflags) $(cvarsmt) $(cdebug) client.c

server.obj: server.c
  $(cc) $(cflags) $(cvarsmt) $(cdebug) server.c
```

ПОСТРОЕНИЕ ГОТОВЫХ К ВЫПОЛНЕНИЮ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ ЯЗЫКА VISUAL C++

Создание файла нового проекта (Project) и построение готовых к выполнению программ

Для создания новой управляющей системы на языке Visual C++ используйте шаги, приведенные в примере ECHO. Ниже приводится список шагов:

- Для создания новой управляющей системы используйте выбор пункта меню Project New.
- Определите управляющую систему как Console Application этот пример не под Windows 3.1.
- Выберите файл Server.c для добавления к проекту.
- Установите опции Проекта (Project), используя выбор пункта меню Options Project.
- В Linker Options не нужны никакие изменения.

Новое в MULTI

Внесите следующие изменения в Compiler Options.

— Выберите опции Runtime.

— Щелкните кнопкой мыши по опции Multi Threaded (LIBCMT.LIB).

- Постройте готовую к выполнению программу, щелкнув мышью по клавише *Build.*
- Установите аргументы командной строки, используя пункт Options Debug: **port 100 0**.
- Выполните те же шаги, что и выше, для добавления Client.c к другому проекту и постройте готовую к выполнению программу.
- Установите параметры командной строки в **\\.\pipe\port100**

ПРИМЕРЫ ПРОТОКОЛОВ

```
Calling CreateNamedPipe('\\.\pipe\brian'...)
  MessageMode:Byte
  PipeMode  :Byte
  WaitMode  :Wait
Calling CreateNamedPipe to create a new pipe instance
Created Pipe instance 100
Calling ConnectNamedPipe on pipe 100 to accept a new client
A Client is available
```

Клиентом является Client 1, который ожидает сервер.

```
Calling CreateNamedPipe to create a new pipe instance

Created Pipe instance 96
Calling ConnectNamedPipe on pipe 96 to accept a new client
A Client is available
```

Клиентом является Client 2, который только что начал работу.

```
Calling CreateNamedPipe to create a new pipe instance
Created Pipe instance 88
Calling ConnectNamedPipe on pipe 88 to accept a new client
A Client is available
```

Клиентом является Client 3, который только что начал работу.

```
Calling CreateNamedPipe to create a new pipe instance
The Maximum number of clients have been reached
Main Server sleeping for 10 Seconds
```

В наличии имеется всего четыре цепочки, три для клиентов и одна для главного сервера.

```
Client 'one' has connected to pipe 100
```

Это сообщение пришло от цепочки Client 1.

```
Client 'two' has connected to pipe 96
```

Это сообщение пришло от цепочки Client 2.

```
Calling CreateNamedPipe to create a new pipe instance
The Maximum number of clients have been reached
Main Server sleeping for 10 Seconds
```

Главный сервер все еще находится в режиме ожидания (sleeping), т.к. в наличии имеется три программных канала и он пытается создать четвертый.

```
Client 'three' has connected to pipe 88
```

Сообщение пришло от цепочки Client 3.

```
Calling CreateNamedPipe to create a new pipe instance
The Maximum number of clients have been reached
Main Server sleeping for 10 Seconds
```

Главный сервер все еще находится в режиме ожидания, т.к. в наличии имеется три программных канала и он пытается создать четвертый.

```
one:Hello from one
two:Hello from two
Calling CreateNamedPipe to create a new pipe instance
The Maximum number of clients have been reached
Main Server sleeping for 10 Seconds
```

Главный сервер все еще находится в режиме ожидания, т.к. в наличии имеется три канала и он пытается создать четвертый.

```
three:Hello from three
Calling CreateNamedPipe to create a new pipe instance
The Maximum number of clients have been reached
Main Server sleeping for 10 Seconds
```

Главный сервер все еще находится в режиме ожидания, т.к. в наличии имеется три канала и он пытается создать четвертый.

```
one:bye from one
received 'bye' command on client [one], client being disconnected
```

Цепочка 1 только что вышла (из цикла) и закрыла свой описатель программного канала. Так как программный канал больше не существует, то после выхода главной цепочки из режима ожидания она способна создать программный канал.

```
Calling CreateNamedPipe to create a new pipe instance
Created Pipe instance 100
```

Наконец программный канал создан.

```
Calling ConnectNamedPipe on pipe 100 to accept a new client
A Client is available
```

Это Client 4, который ожидал в течение определенного времени.

```
Calling CreateNamedPipe to create a new pipe instance
The Maximum number of clients have been reached
Main Server sleeping for 10 Seconds
```

После того как Client 4 был принят, количество экземпляров снова стало равно трем, поэтому попытка создать еще один терпит неудачу.

```
Client 'four' has connected to pipe 100
four:Hello from four
Calling CreateNamedPipe to create a new pipe instance
The Maximum number of clients have been reached
Main Server sleeping for 10 Seconds
```

Главный сервер все еще находится в режиме ожидания, так как в наличии имеется три канала и он пытается создать четвертый.

```
two:exit from two
received 'exit' command on client [two]
```

Теперь, когда два ушли, вы можете создать ваш новый экземпляр, когда через 10 секунд ожидания главная цепочка "просыпается".

```
Calling CreateNamedPipe to create a new pipe instance
Created Pipe instance 96
Calling ConnectNamedPipe on pipe 96 to accept a new client
```

Главная цепочка блокируется внутри *ConnectNamedPipe*, ожидая подключения нового клиента.

```
three:quit from three
received 'quit' command on client [three], Server Exiting
```

Команда quit осуществляет ExitProcess (завершение процесса), поэтому все цепочки переходят в режим завершения и процессы прекращаются.

ПРОТОКОЛ КЛИЕНТА 1 (CLIENT 1)

```
Calling WaitNamedPipe to see if a pipe is available
The Pipe does not exist
```

Это происходит потому, что клиент 1 стартовал раньше сервера. Ввиду того что сервер не работает, программного канала просто не существует.

```
Sleeping for a second
Awake!
Calling WaitNamedPipe to see if a pipe is available
A Pipe instance is available
```

Теперь сервер стартовал.

```
Calling CreateFile to connect to the pipe
```

Клиент подключен!

```
The first line of data you send will become your Client id Enter data to
send to the server, exit to quit
Client>one
[4] characters transmitted
Client>Hello from one
[15] characters transmitted
Client>bye from one
[13] characters transmitted
Client>bye
[4] characters transmitted
Client>more data???
Error:232
The pipe is being closed.
```

ПРОТОКОЛ КЛИЕНТА 2 (CLIENT 2)

```
Calling WaitNamedPipe to see if a pipe is available
A Pipe instance is available
Calling CreateFile to connect to the pipe
The first line of data you send will become your Client id
Enter data to send to the server, exit to quit
Client>two
[4] characters transmitted
Client>Hello from two
[15] characters transmitted
Client>exit from two
[14] characters transmitted
Client>exit
```

```
[5] characters transmitted
Pipe being closed
```

ПРОТОКОЛ КЛИЕНТА 3 (CLIENT 3)

```
Calling WaitNamedPipe to see if a pipe is available
A Pipe instance is available
Calling CreateFile to connect to the pipe
The first line of data you send will become your Client id
Enter data to send to the server, exit to quit
Client>three
[6] characters transmitted
Client>hello from three
[17] characters transmitted
Client>quit from three
[16] characters transmitted
Client>quit
[5] characters transmitted
Client>more data???
The pipe is no longer connected
Client exiting...
```

ПРОТОКОЛ КЛИЕНТА 4 (CLIENT 4)

```
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
```

Сервер уже общается с тремя клиентами, поэтому этот должен подождать. Когда клиент обращается к функции *WaitNamedPipe*, он может уточнить, как долго продлится тайм-аут. Клиент уточнил величину NMPWAIT_USE_DEFAULT_WAIT, что означает использование такого количества времени, которое было указано сервером, когда он создавал этот программный канал. Эта величина равнялась 10 000 миллисекундам, или 10 секундам.

```
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
Timeout...TryAgain
Calling WaitNamedPipe to see if a pipe is available
A Pipe instance is available
```

Клиент 1 окончательно вышел из цикла.

```
Calling CreateFile to connect to the pipe
The first line of data you send will become your Client id
```

```
Enter data to send to the server, exit to quit
Client>four
[5] characters transmitted
Client>hello from four
[16] characters transmitted
```

Теперь Клиент 3 передает команду на завершение сеанса, что ведет к выходу из цикла сервера. Ошибка 232 происходит тогда, когда клиент в следующий раз попытается ввести информацию в программный канал.

```
Client>more data???
Error:232
The pipe is being closed.
```

Глава 9

Реализация программы на языке VISUAL C++

ВВЕДЕНИЕ

Предназначение

Цель данного примера — показать, как следует реализовать "с листа" программу на Visual C++ фирмы Microsoft. Программа представляет собой графический интерфейс пользователя (Graphical User Interface — GUI) с сервером гнезда. Вместо выполнения программой всех шагов сервера пользователь выполняет различные шаги путем выбора пунктов меню. На каждом шагу пользователь имеет возможность просмотреть различные гнезда, имеющиеся в системе. Данная программа была написана и отлажена на машине под Windows NT, но может быть адаптирована для машины под Windows/DOS.

Клиент

В программе не содержится клиентской части. После запуска сервера вы можете подключить к нему любого клиента. От пользователя прикладной программы зависит передача и получение соответствующих команд от клиента.

Сервер

Данный пример настолько сложен, что, прежде чем анализировать код программы, лучше рассмотреть рабочий пример сервера и клиента.

ЦИКЛ СЕРВЕРА

```
D:\BOOK\ECHO\SERVER.EXE
Server Echo Program 1.0

        Calling WSAStartup(Version,Details) to locate a Windows DLL
        WSAStartup has returned ImplementationDetails:
          wVersion:      (257)
          wHighVersion: (257)
          szDescription:'Microsoft Windows Sockets Version 1.1.'
          szSytemStatus:'Running.'
          iMaxSockets:  (32767)
          iMaxUdpDg:    (65467)
          lpVendorInfo  (00014C6C)


        Calling socket(AddressFormat,Type,Protocol) to Create a Server Socket
        Server Socket (92) has been created

        Calling bind(Socket,AddressPointer,length) to Bind the socket
        Bind succeeded

        Calling listen(Socket,MaxConnections)
        Successfully changed to a listen socket

        Calling accept(ListenSocket,AddressPointer,length)
        This function will BLOCK until a client is available
```

Рис. 9-1. Экран сервера.

Меню SocketFunctions используется для выполнения различных функций, которые должен реализовывать ваш сервер. Вывод на дисплей вызванной функции появляется в области клиента — пустой области экрана (см. рис. 9-1) — окна прикладных программ сервера.

Пункт меню fdSets используется для проверки четырех наборов дескрипторов файла (fd_sets), которые определены в сервере. Набор fd_set *AllSockets* содержит все те гнезда, которые определены для системы. Три других fd_sets содержат либо те же самые значения, что и *AllSockets*, при условии их инициализации, — либо они содержат гнезда с подходящим набором свойств. Например, *ReadSockets* представляет собой список гнезд, которые содержат данные, готовые к считыванию.

Пункт меню Actions используется для выполнения специальных действий в сети, таких, как передача и прием данных.

Меню Flags используется для изменения значения флага на противоположное.

Пункт меню Help не используется, за исключением режима по умолчанию, созданного с помощью Visual C++.

ФУНКЦИИ ГНЕЗД

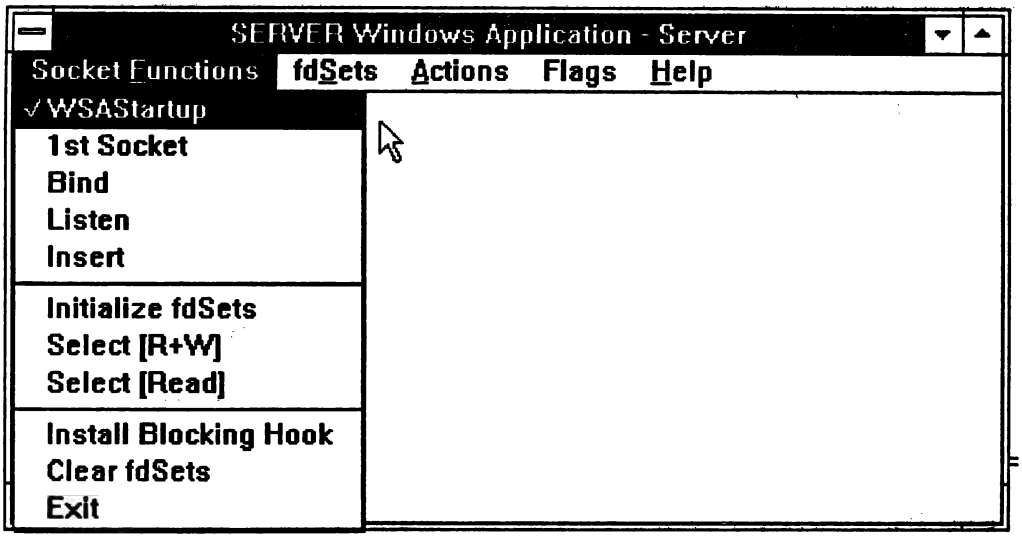


Рис. 9-2. Меню Socket Functions (Функции гнезд).

В пункте меню Socket Functions (см. рис. 9-2) вы найдете различные функции, к которым должен обращаться ваш сервер. Программа путем установки контрольной метки (a check mark) ("галочки") рядом с пунктом дает вам подсказку для выбора наиболее подходящей функции. В данном примере указывается на то, что вы должны обратиться к WSAStartup для инициализации сети.

Совсем необязательно следовать подсказке. Однако, если вы поступаете таким образом, последующие подсказки могут быть ошибочными, т.к. они отмечают вызов функции, который вы только что сделали, и помечают логически следующий после этого пункт меню.

Если вы хотите исследовать то, что произошло, когда вы не обращались к WSAStartup, или вы обратились к функции вне очереди, данный сервер позволит вам просмотреть допущенные вами ошибки. В этом примере, однако, мы придерживаемся подсказок.

Если вы выберете пункт меню WSAStartup, вы получите на дисплее картинку, изображенную на рис. 9-3.

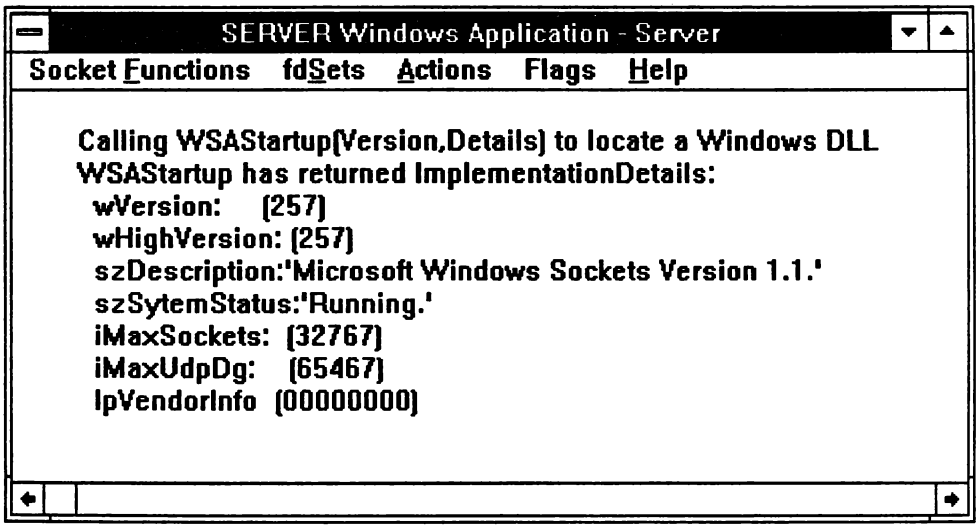


Рис. 9-3. Отображение пункта WSA Startup.

Протокольная информация, которая обычно распечатывается на стандартном устройстве вывода, отображается в клиентской области окна. Меню Socket Functions теперь выглядит так, как показано на рис. 9-4.

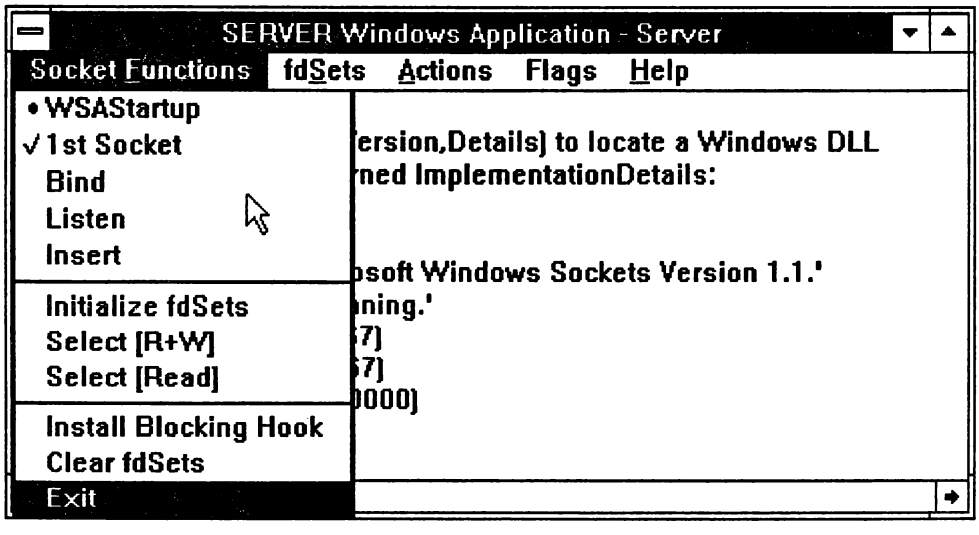


Рис. 9-4. Меню Socket Functions изменяется для обозначения следующего пункта для выбора.

Заметьте, что перед пунктом WSAStartup теперь стоит точка, а пункт меню st Socket теперь помечен галочкой. Это означает, что вы уже выполнили WSAStartup. Пункт меню 1st Socket создает первое гнездо, которое используется как гнездо сервера. После того как вы выбрали пункт меню, на экране появляется информация, изображенная на рис. 9-5.

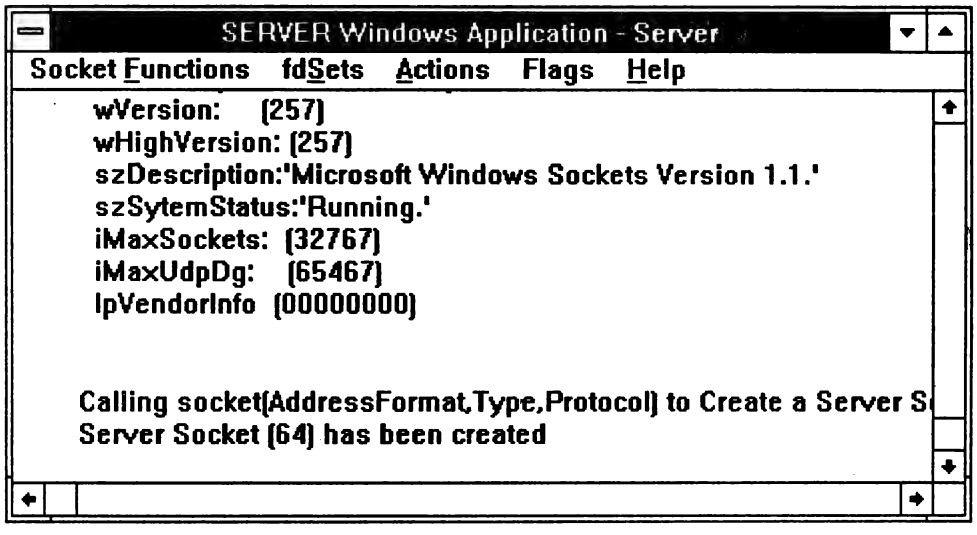


Рис. 9-5. Вид экрана после выбора пункта меню 1st Socket.

Теперь, когда текст не помещается в пределы окна, вы обнаруживаете автоматическое появление линеек прокрутки, и таким образом вы имеете возможность просмотреть файл протокола в окне в разных направлениях.

Следующий пункт меню для выбора — функция Bind. После выбора вами данной функции на экране появляется окно диалога, показанное на рис. 9-6:

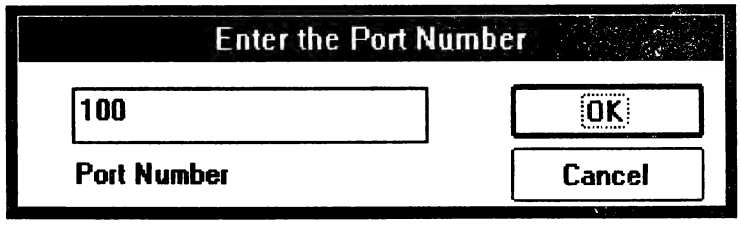


Рис. 9-6. Диалоговое окно после выбора функции Bind.

Порт 100 используется по умолчанию. При нажатии клавиши возврата <Return> гнездо привязывается к порту 100 на текущем узле.

Следующей функцией для выполнения является функция Listen. Она используется для переключения только что созданного гнезда в гнездо прослушивания. После выбора этого пункта меню на экране появляется картинка, показанная на рис. 9-10.

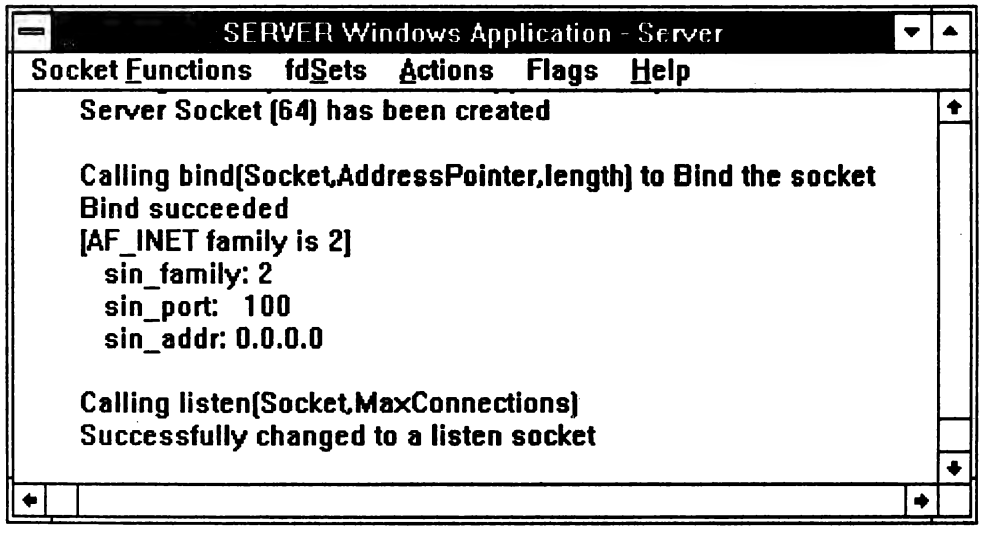


Рис. 9-10. Экран после выбора Listen.

Теперь пункт меню Socket Functions выглядит, как на рис. 9-11.

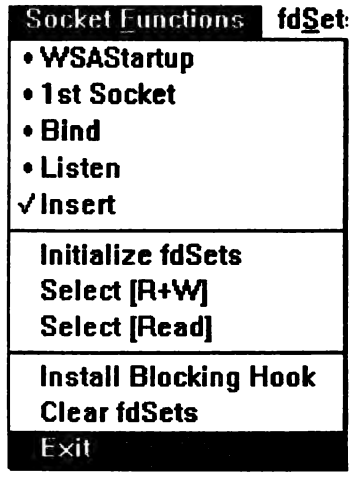


Рис. 9-11. Меню Socket Functions.

Следующим логическим шагом является вставление (Insert) гнезда, которое вы создали, привязали и превратили в гнездо прослушивания в fd_set *AllSockets*. Если вы изучите пункт меню fdSets в этом месте, вы увидите, что все fd_sets в настоящий момент пусты. Это проверяется попыткой вытащить содержимое пункта меню All Sockets. Пункт меню не содержит каких-либо подпунктов (см. рис. 9-12).

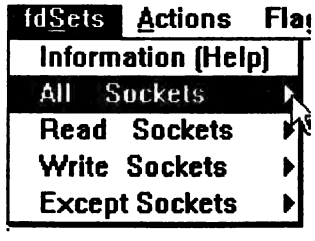


Рис. 9-12. Пункт меню All Sockets.

После выбора пункта меню Insert экран отражает информацию, которую можно найти в меню fdSets (см. рис. 9-13).

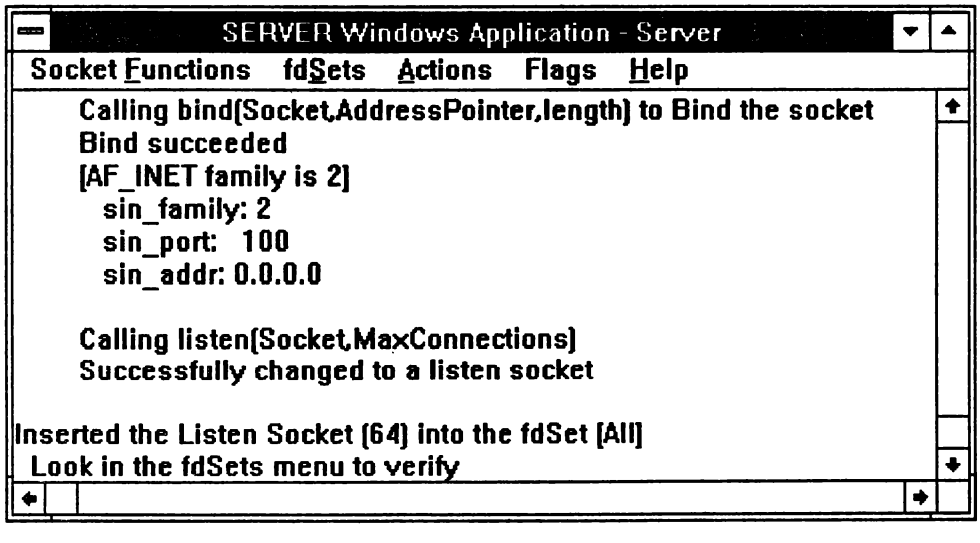


Рис. 9-13. Подсказка относительно меню fdSets.

Теперь гнездо fd_set All Sockets работает, как показано на рис. 9-14.

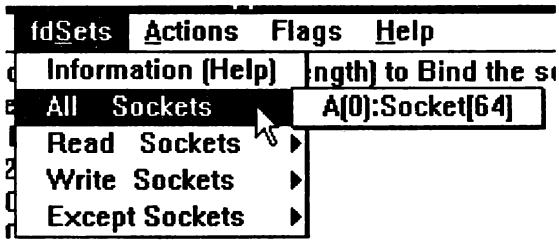


Рис. 9-14. Набор fd_set All Sockets.

Меню Socket Functions теперь показывает, что вы должны инициализировать *fd_sets* (см. рис. 9-15). Данная операция копирует текущее содержание *All Sockets* в три другие *fd_sets*. В данный момент три другие *fd_sets* пусты:

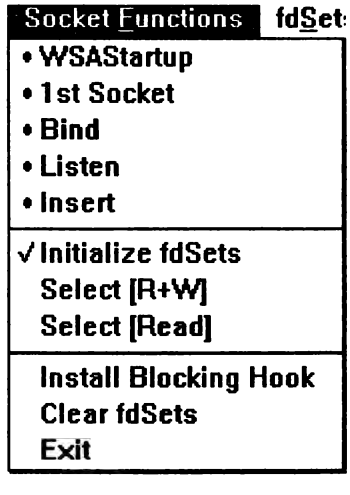


Рис. 9-15. Инициализация fdSets.

Теперь, после инициализации *fd_sets*, вы можете обратиться к функции Select. Существуют две варианта функции Select. В первом варианте выборка ведется по всем трем fd_sets: считывание, запись и исключение (Except). Этот вариант редко блокируется, т.к. большую часть времени по крайней мере одно гнездо доступно для записи. Второй вариант функции Select переходит в NULL для записи fd_set. Он блокируется до того момента, пока не появится какой-либо тип активности считывания. Рис. 9-16 отображает реализацию выбора функции Select [Read]:

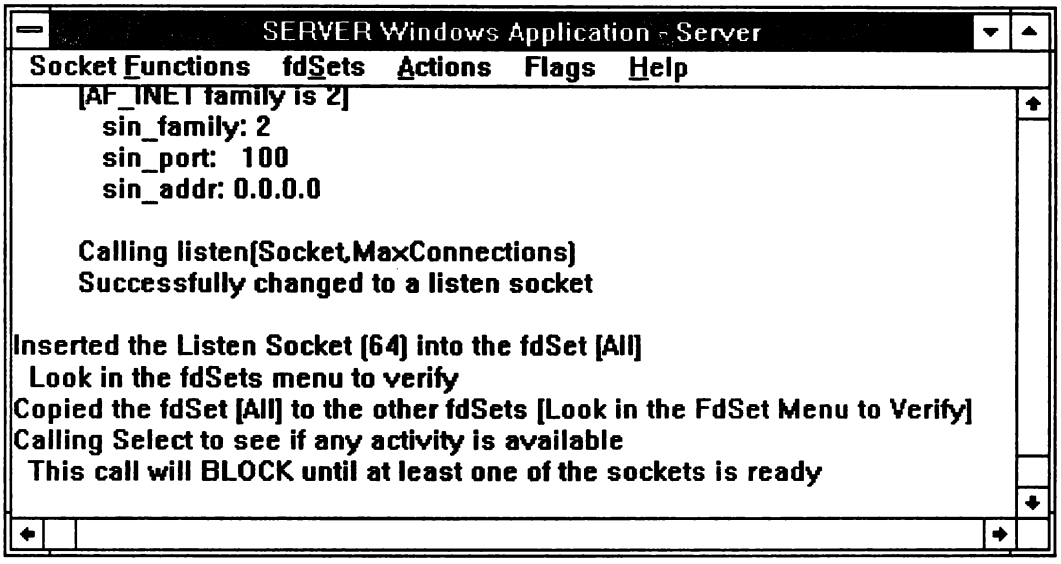


Рис. 9-16. Результат выбора функции Select [Read].

Единственная серьезная проблема, связанная с данной функцией Select, является ее блокирование. Когда функция блокируется, цикл обработки запроса для окна уже не может осуществлять обработку запросов для окна сервера. Вы можете проверить это, попытавшись передвинуть окно. Если другая прикладная программа занимает экран, занятый окном сервера, и потом снова открывает область, то сервер блокируется в функции Select это означает, что он не может вновь вывести изображение на эту область экрана.

Такой режим нежелателен для Windows NT и неприемлем для Windows 3.1. В Windows 3.1 ни одна из прикладных программ не может быть выполнена, пока не произойдет возврата функции Select.

К счастью, библиотека WINSOCK обладает механизмом по выходу из этой ситуации. Это специальная функция обратного вызова, которая называется *Blocking Hook* (ловушка блокирования). После установки функции ловушки блокирования она принимает сообщения в окне до тех пор, пока сервер заблокирован. Функция ловушки блокирования позволяет вам обеспечивать нормальную работу окна, пока сервер заблокирован в функциях Select. Ловушка блокирования отключает все пункты меню, которые мешают выполнению функции Select.

Используя перечень задач для Windows NT, выберите прикладную программу сервера и завершите ее. Это в настоящий момент самый легкий путь выхода из заблокированного состояния. Другим способом выхода из заблокированного состояния является запуск модуля пользователя и попытка его подключения к серверу. Как только будет сделан запрос на подключение, функция Select возвратится и сервер разблокируется.

Прикладная программа сервера запускается вновь, и еще до вызова функции Select [Read] происходит обращение к функции Install Blocking Hook и активизируется

функция Select [Read]. На рис. 9-17 показано меню Socket Functions (Функции гнезда) в тот момент, когда функция Select является блокирующей при установленной ловушке блокирования.

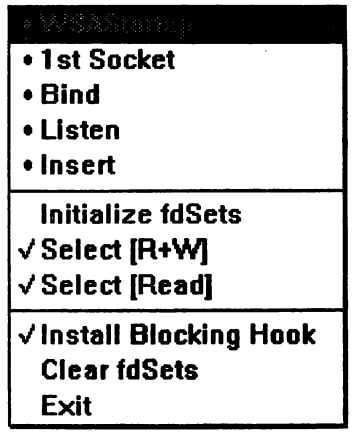


Рис. 9-17. Меню с установленной ловушкой блокирования.

Следует заметить, что большинство функций отключено. Вы обнаружите, что любой пункт меню, вызывающий функцию гнезда или имеющий дело с гнездами, будет отключен. Галочка, стоящая перед пунктом меню Install Blocking Hook (Установить ловушку блокирования), указывает на то, что ловушка блокирования в данный момент установлена. Повторный вызов данного пункта меню отключает ловушку блокирования и возвращает вас обратно к нежелательному режиму, блокирующему работу всех окон сервера.

Стандартный клиент ECHO запускается и подключается к серверу. Экран сервера выглядит следующим образом:

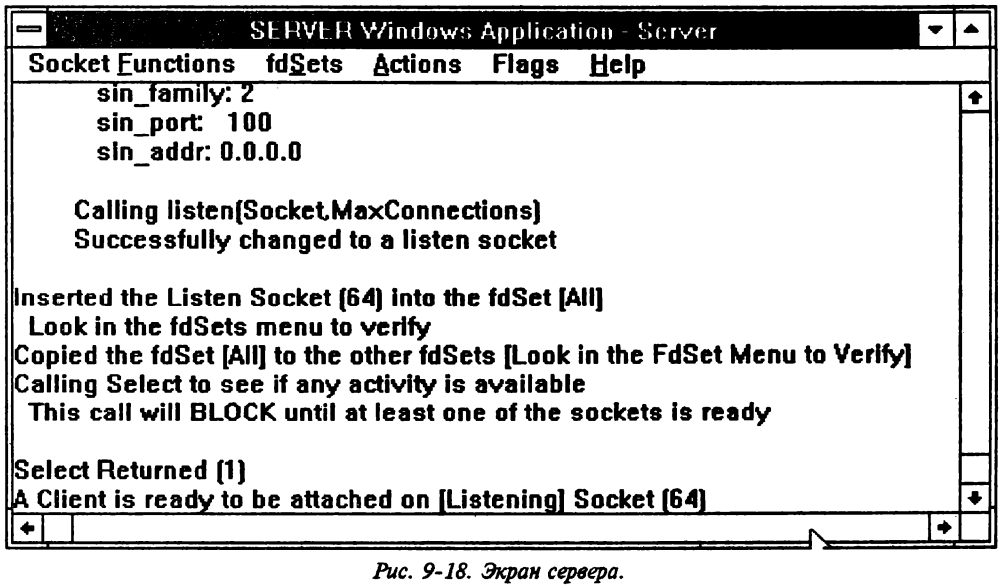


Рис. 9-18. Экран сервера.

Функция Select возвратила 1, указывающую на задействование одного гнезда. После изучения данных она определила, что клиент готов к подключению.

Клиент может быть подключен следующим образом:

- Путем выбора гнезда из меню Read Sockets fd_set, которое теперь содержит гнездо с доступными данными.
- Путем выбора функции Accept из меню Actions, которое теперь содержит галочку, обозначающую, что она является подходящим видом активности.
- Путем выбора функции Accept, вызывающей появление на экране картинки, изображенной на рис. 9-19.

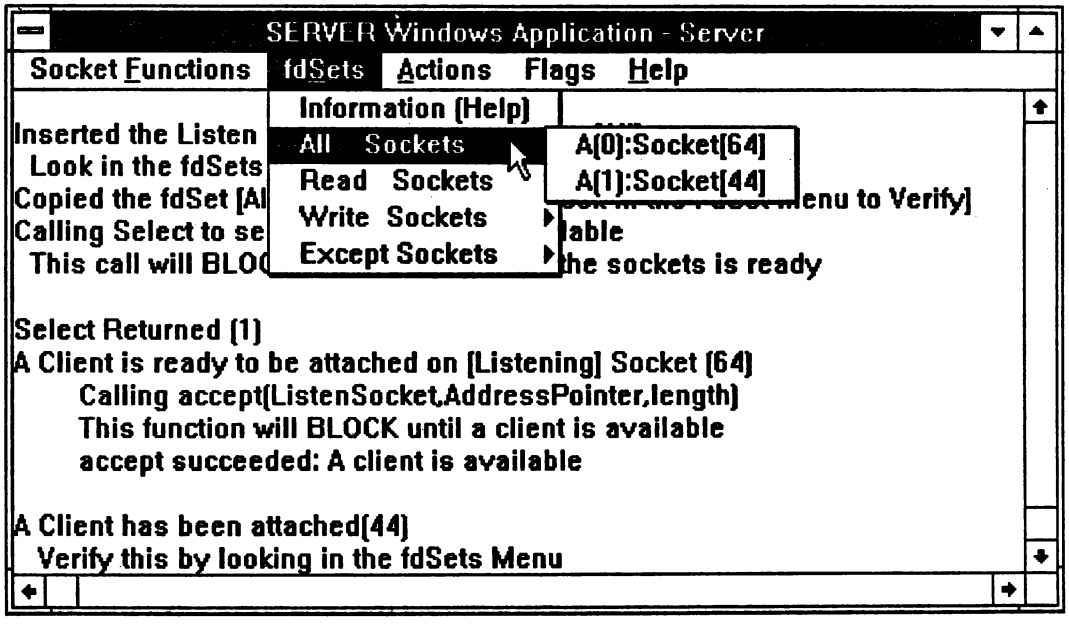


Рис. 9-19. Выбор функции Accept.

Следует заметить, что новое гнездо вставлено в fd_set *AllSockets*.

Теперь вы можете запустить снова *fd_sets* для копирования *AllSockets* в другие fd_sets, а после этого вы можете вызвать Select [Read].

Если вы забыли обратиться к Initialize, то, как правило, Select немедленно возвратится со значением 0. Это происходит потому, что fd_sets пусты; таким образом, функция Select осуществляет выбор без гнезд. Если вы проделали это, просто вызовите Initialize и вновь активизируйте Select.

Рисунок 9-20 создается следующим образом:

- путем вызова Initialize;
- путем вызова Select [Read];
- путем набора данных на модуле клиента и нажатия <RETURN>;
- путем выбора R[0]: Гнездо [44] из **fd_set** Read.

```
SERVER Windows Application - Server
Socket Functions   fdSets   Actions   Flags   Help
A Client is ready to be attached on [Listening] Socket [64]
      Calling accept(ListenSocket,AddressPointer,length)
      This function will BLOCK until a client is available
      accept succeeded: A client is available

A Client has been attached(44)
  Verify this by looking in the fdSets Menu
Copied the fdSet [All] to the other fdSets [Look in the FdSet Menu to Verify]
Calling Select to see if any activity is available
 This call will BLOCK until at least one of the sockets is ready

Select Returned (1)
Data Ready on Socket (44)
      Calling recv(Socket,buffer,SizeOfBuffer,Flags)
socket:044>[018]Hello from client\000
```

Рис. 9-20. Клиент заблокирован, ожидая, пока сервер отправит сообщение.

Теперь Клиент заблокирован, ожидая, пока сервер отправит сообщение обратно. Помните, что сервер ECHO добавил 1 к каждому символу и передал данные обратно клиенту. Клиент теперь ожидает прихода данных. В данном примере вы просто посылаете сообщение обратно клиенту.

Для передачи данных на гнездо:

- Инициализируйте fd_sets.
- Вызовите Select [R+W].
- Select должен возвратить значение 1, обозначающее, что существует одно активное гнездо — оно находится в fd_set Write (Запись).
- Щелкните мышью по гнезду или выберите опцию Send в меню Actions.

После выполнения этих шагов вы получите окно, приведенное на рис. 9-21.

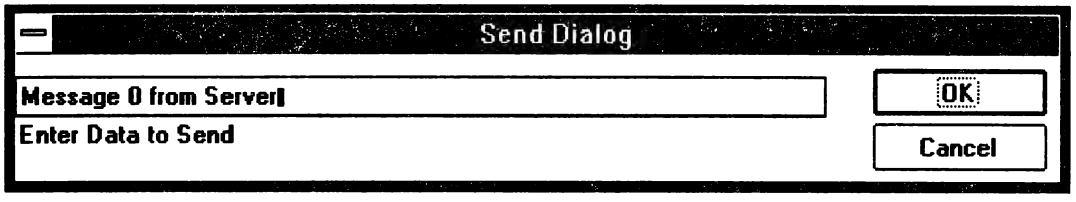


Рис. 9-21. Диалоговое окно Send (Отправить).

Это — диалоговое окно, которое позволяет вам отправить сообщение. Вы можете принять сообщение по умолчанию, или вы можете изменить сообщение до того, как вы его отправите.

Щелкнув по клавише OK, вы активизируете следующую картинку в Сервере и разблокируете клиента.

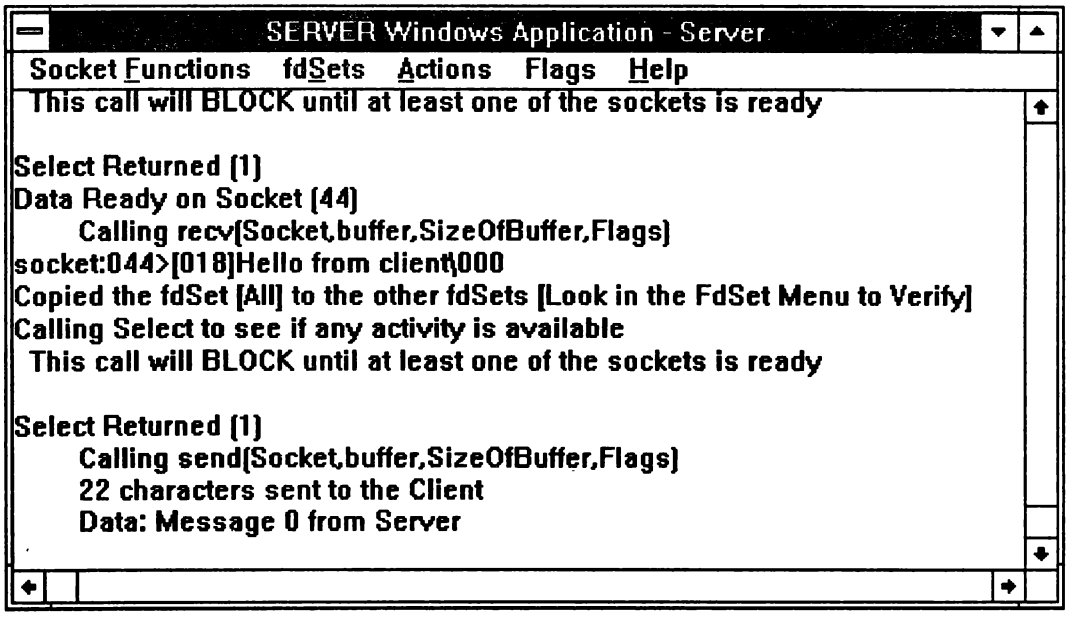


Рис. 9-22. Разблокирование клиента.

Следующие шаги позволяют осуществить основной процесс:

- Инициализация набора fd_sets.
- Выбор [Read] или Select [R+W].
- Для активизации гнезд щелкните по ним кнопкой мыши.

СОЗДАНИЕ ПРОЕКТА СЕРВЕРА

Введение

Теперь, когда вы уже знакомы с порядком работы сервера, мы рассмотрим процедуру, используемую для создания этой прикладной программы в среде языка Visual C++ фирмы Microsoft. Вы можете начать с систем Windows NT или Windows 3.1. Однако, запустив процесс однажды, вы не сможете переключать код программы с одной платформы на другую. Это происходит потому, что используемые модули оперативной помощи прикладных программ на языке Visual C++ имеют некоторый специфический код для платформы. Если вы хотите переключиться на другую платформу, вы должны будете начать процесс с самого начала.

В данном примере используется язык Visual C++ редакции V1.10 на 32 бита для Windows NT.

AppWizard

После запуска Visual C++ выберите пункт AppWizard из меню Project. Появится диалоговое окно, приведенное на рис. 9-23.

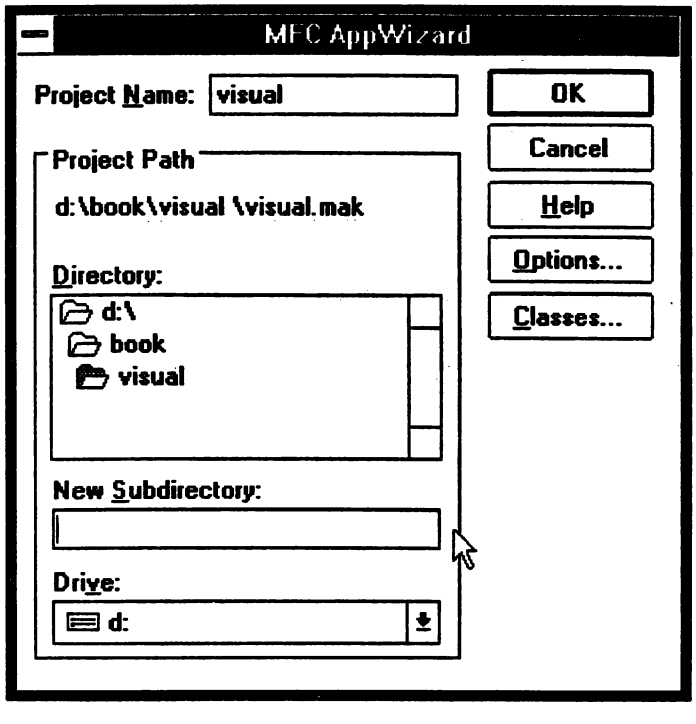


Рис. 9-23. Диалоговое окно AppWizard.

Этот проект будет называться в дальнейшем Visual и находится в определенном каталоге. Так как каталог уже существовал, субкаталог New Subdirectory, создать который предложил AppWizard, был исключен.

Опции AppWizard

Следующие опции должны быть отключены путем выбора кнопки опций и щелчка кнопкой мыши по следующим триггерным кнопкам для их отключения:

- Multiple Document Interface
- Initial Toolbar
- Printing and Print Preview

После этих изменений картина на дисплее будет выглядеть так, как показано на рис. 9-24.

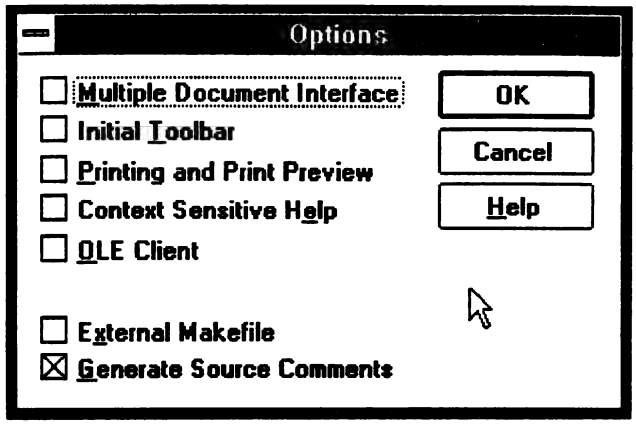


Рис. 9-24. Отключенные опции AppWizard.

Щелкните кнопкой мыши по клавише OK для принятия этих изменений.

Щелкните по OK в диалоге AppWizard для создания проекта. Так как вы не добавляете какие-либо специальные классы к проекту, то вам нет необходимости выходить в диалоговое окно Classes....

Диалоговое окно, которое подтверждает ваши варианты выбора, показано на рис. 9-25.

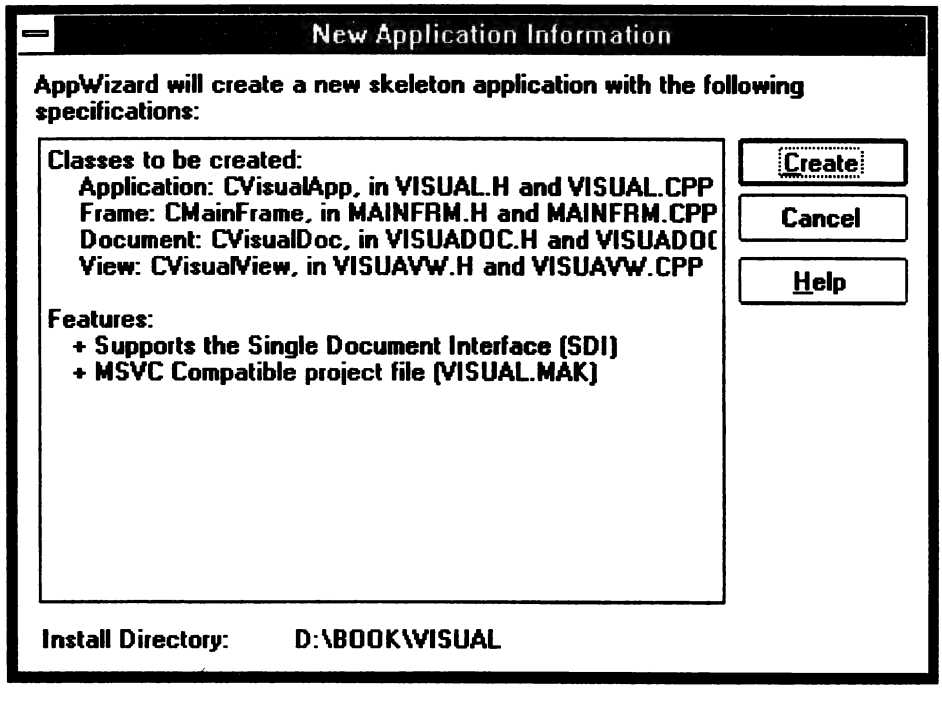


Рис. 9-25. Диалоговое окно New Application Information.

Щелкните по клавише Create для создания проекта.

МОДИФИКАЦИЯ МЕНЮ

App Studio

Используйте опцию App Studio в меню Tools для активизации прикладной программы App Studio с тем, чтобы отредактировать меню, пиктограммы, диалоговые окна и другие ресурсы. Появляющееся при этом окно показано на рис. 9-26.

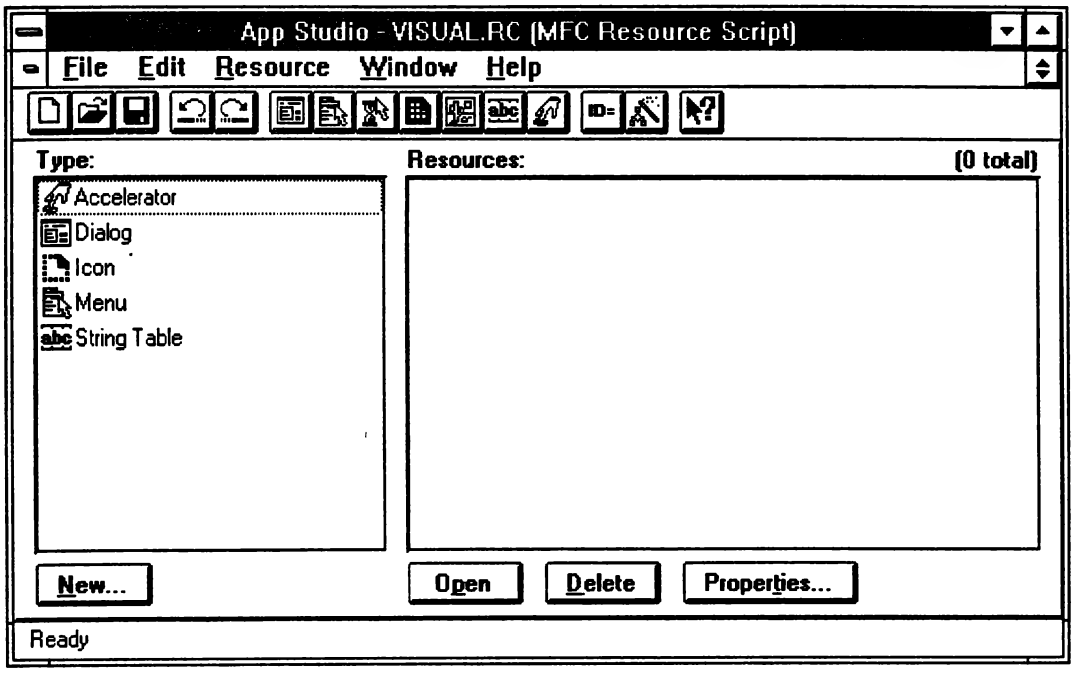


Рис. 9-26. Главное окно прикладной программы App Studio.

Щелкните кнопкой мыши по пиктограмме Menu для вывода на экран меню ресурсов в вашем проекте.

Для активизации редактора меню необходимо дважды щелкнуть кнопкой мыши по единственному ресурсу меню, находящемуся в данный момент в вашем проекте. Он расположен в правом подокне окна.

Редактирование файлового меню

Щелкните кнопкой мыши по меню File в области клиента для изменения файлового меню. Двойной щелчок по меню File приводит к активизации диалогового окна Properties, как показано на рис. 9-27.

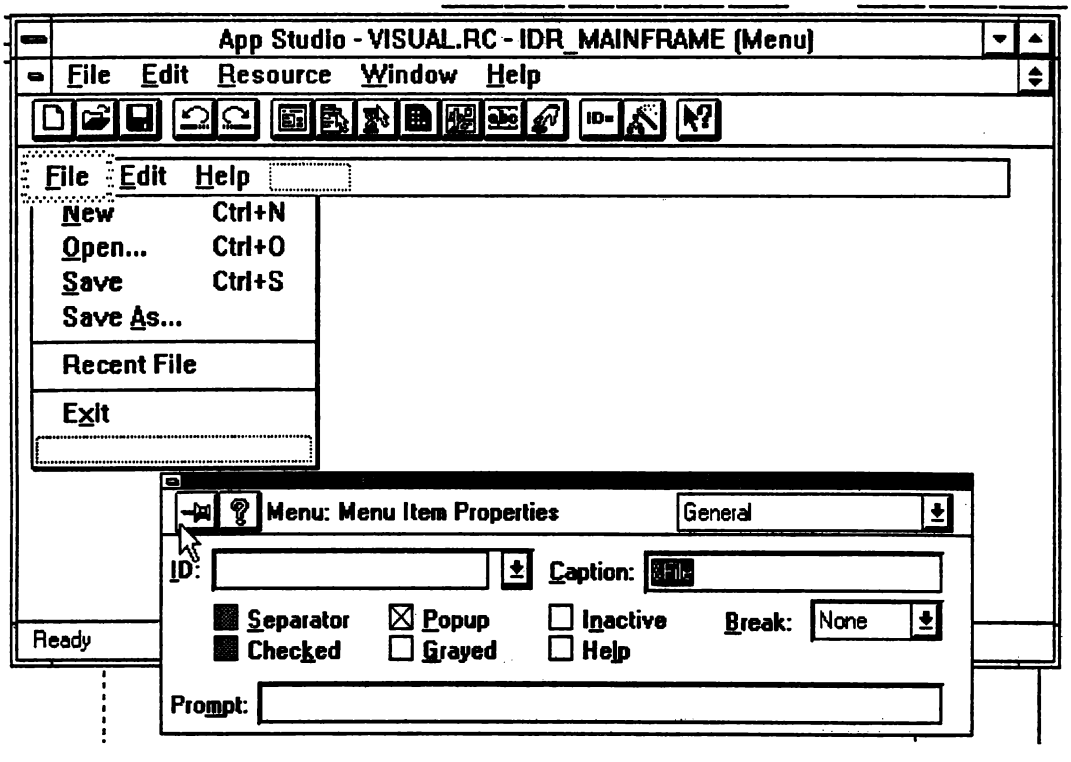


Рис. 9-27. Диалоговое окно Properties (свойства).

Меню Item Properties содержит пиктограмму *thumb tack* (*палец на клавише*), которая используется для превращения диалогового окна в немодальное. Щелкните по этой клавише в диалоговом окне Item Properties. Стрелка указывает на клавишу в диаграмме. Теперь диалоговое окно будет привязано к вашему экрану, пока вы вносите изменения в различные части меню.

Используйте Item Properties для изменения заголовка с &File на Socket & Functions. Затем внесите следующие изменения:

This Option... becomes this new value...	Данная опция ... становится этим новым значением ...
New WSAStartup	
ID_FILE_NEW	ID_WSAStartup
prompt...	Initialize the Windows Socket system

Open 1st Socket
ID_FILE_OPEN ID_Socket
prompt... Create the Primary Socket

Save Bind
ID_FILE_SAVE ID_Bind
prompt... Bind the primary socket to a network
address and port

Save As Listen
ID_FILE_SAVE_AS ID_Listen
prompt... Change the Primary socket to a
Listen Socket

Recent File Initialize fdSets
ID_FILE_MRU_FILE1 ID_Initialize
prompt... Copy AllSocket to other fdSets
[greyed attribute] [not greyed]

Клавиша <Insert> вставит пустой пункт меню перед выделенным в данный момент пунктом. Вставьте следующий пункт после пункта Listen перед разделителем:

Insert
ID_Insert
Insert the Listen Socket into the AllSockets fdSet

Вставьте следующие два пункта после пункта меню Initialize fdSets:

Select [R+W] Select [Read]
ID_Select ID_SelectRead
Подождите, пока данные не будут доступны в гнездах.

Вставьте следующие пункты меню перед командой EXIT (ВЫХОД):

Install Blocking Hook
ID_InstallBH
Install the Blocking Hook

Clear fdSets
ID_ClearfdSets
Clear all the fdSets

Ваш экран теперь примет вид, показанный на рис. 9-28.

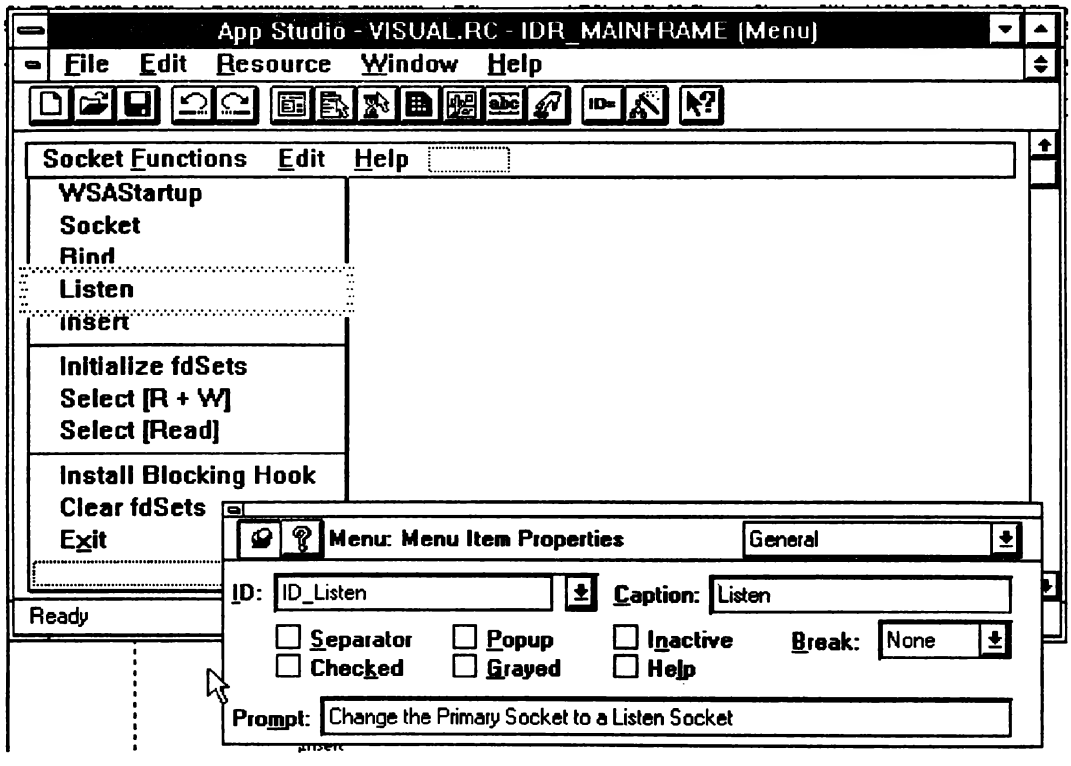


Рис. 9-28. Изменения в меню.

Внесите следующие изменения в меню edit:

This item... Becomes... Данный пункт ... Становится ...

Edit fdSets
Undo Information(Help)
ID_EDIT_UNDO ID_fdSetsHelp
prompt .... prompt ....

Cut All Sockets
ID_EDIT_CUT IF_fdSets
prompt .... prompt ....

Copy Read Sockets
ID_EDIT_CUT ID_fdSets
prompt... prompt ....

Paste Write Sockets
ID_EDIT_PASTE ID_fdSets
Prompt .... prompt ....

```
[new item]  Except Sockets
            ID_fdSets
            prompt ....
```

Щелкните кнопкой мыши по пункту меню Help и нажмите клавишу <Insert> для вставки нового спускающегося меню. Загрузите его следующим:

```
&Actions

Accept      Send        Recv
ID_Accept   ID_Send          ID_Recv
prompt ...  prompt ...       prompt ...

OOB Send        OOB Recv
ID_OOBSend          ID_OOBRecv
prompt ...      prompt ....

separator

Create Connectionless Socket
ID_CreateConnectionlessSocket
prompt ...

Bind Connectionless Socket
ID_BindConnectionlessSocket
prompt ...

Recv From
ID_RecvFrom
prompt ...

Send To
ID_SendTo
prompt ...
```

Создайте новое спускающееся меню под названием Flags.

```
Log
ID_Log
prompt ...
```

Ваше меню должно теперь принять вид, похожий на тот, что изображен на рис. 9-29.

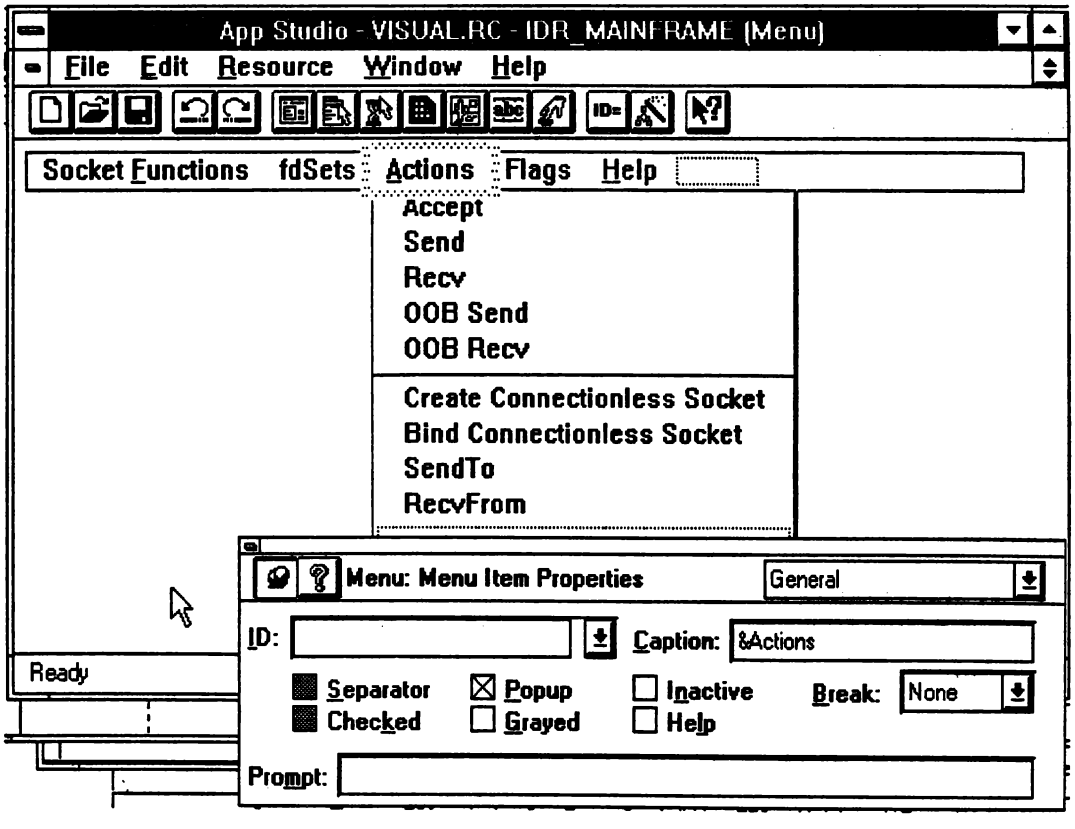


Рис. 9-29. Исправленный экран.

Осуществите выход из App Studio, сохранив все изменения в вашем файле ресурсов.

СОЕДИНЕНИЕ ПУНКТОВ МЕНЮ С КОДОМ ПРОГРАММЫ

ClassWizard (Модули оперативной помощи)

Выберите опцию ClassWizard из пункта меню Browse в Visual C++. Вы получите на экране дисплея картинку, изображенную на рис. 9-30.

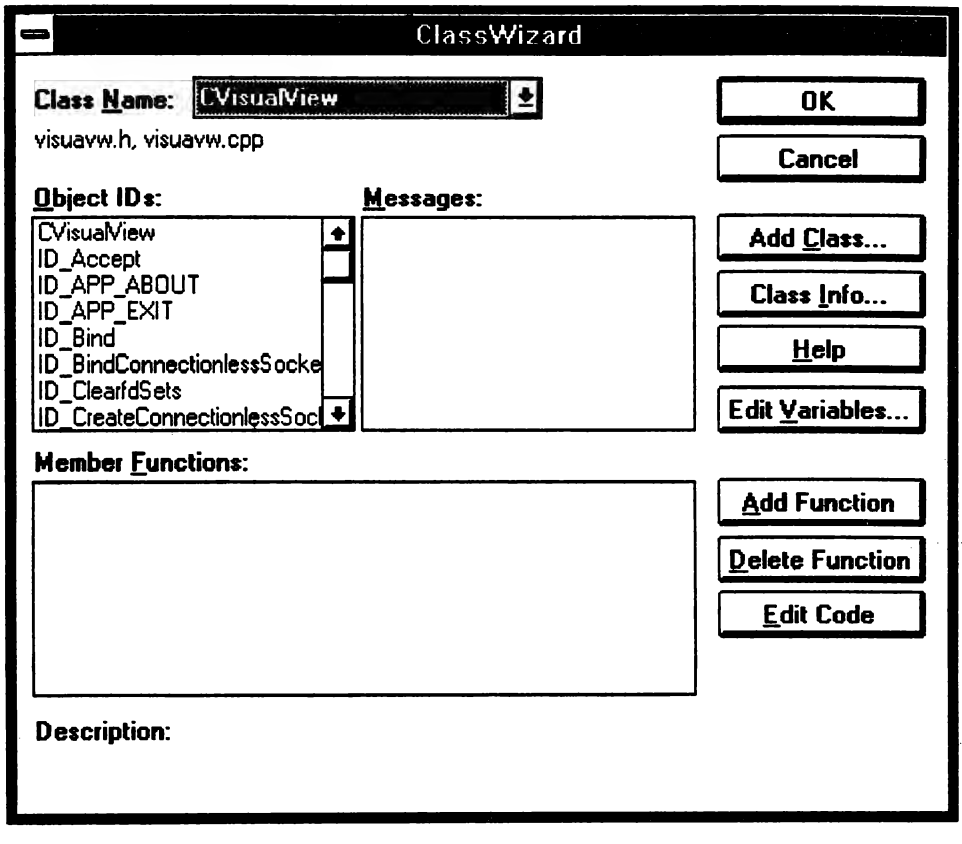


Рис. 9-30. Опция ClassWizard.

Щелкните кнопкой мыши по блоку списка Class Name для просмотра списка классов, которые встроены в вашу прикладную программу, как показано на рис. 9-31.

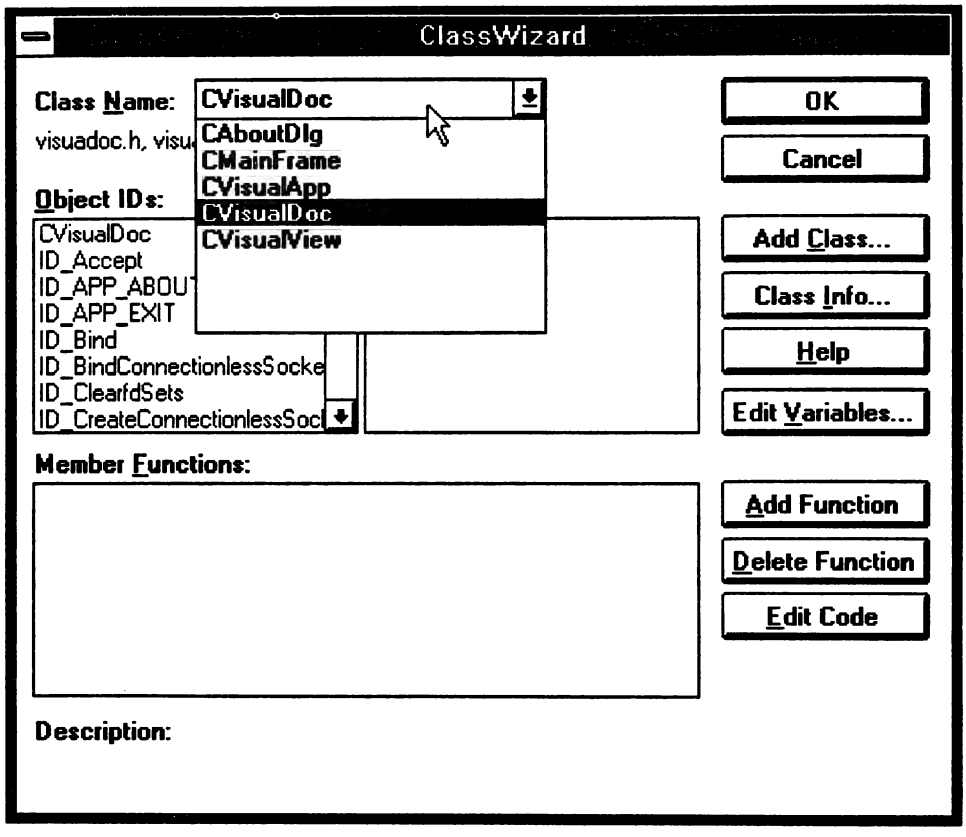


Рис. 9-31. Список встроенных (predefined) классов.

Ваша прикладная программа содержит следующие классы, созданные для вас модулем AppWizard:

- CAboutDlg
  Этот класс является диалоговым классом, который используется для определения прикладной программы, когда пользователь указывает на опцию Help (помощь). Данный класс был сформирован автоматически и не требует выполнения дополнительных операций.

- CMainFrame
  Этот класс является классом Frame (Кадр), который используется для запоминания классов видимого изображения и для реагирования на события, происходящие в окне. Эта прикладная программа содержит единственный Frame Class, т.к. он необходим. Данный класс формируется автоматически и не требует дополнительных операций.

- CVisualApp
  Этот класс является классом прикладных программ, который используется для управления работающей прикладной программой. Данный класс формируется автоматически и не требует дополнительных операций.

- CVisualDoc
  Этот класс является классом документов, который используется для хранения данных прикладных программ. В случае использования данного класса в качестве прикладной программы сервера, он будет запоминать всю регистрируемую информацию. Данный класс является самым приемлемым местом для реагирования на выбор меню и осуществляет большинство операций, связанных с прикладными программами. Данный класс создается автоматически, но требует внесения серьезных изменений для того, чтобы обеспечить функциональные возможности сервера.

- CVisualView
  Этот класс является классом View (Видимое изображение), который используется для обеспечения вывода на экран данных в кадре. Данный класс тесно связан с классом документов для отыскания данных, которые должны выводиться на экран дисплея. Данный класс создается автоматически, но требует внесения некоторых изменений для полного обеспечения функциональных возможностей дисплея.

Модуль оперативной помощи класса (class wizard) используется для соединения кода программы с конкретными идентификаторами (ID). Если вы хотите записать программу обработки для конкретного пункта меню, вы выбираете ID для этого пункта меню из левого подокна, а пункт COMMAND из правого подокна. Компонентная функция (member function), связанная с этой операцией, будет выделена. Если ни одна из функций не определена, то вы можете создать какую-либо функцию, щелкнув кнопкой мыши по клавише Add Function (см. рис. 9-32).

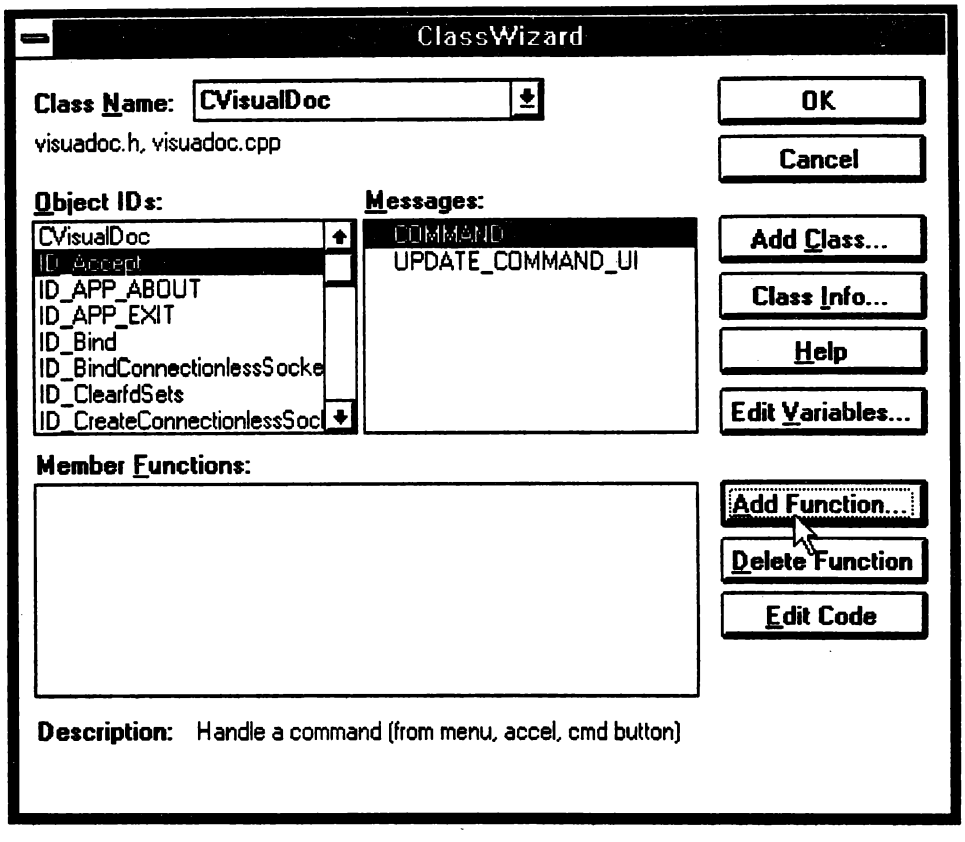


Рис. 9-32. Использование ClassWizard для соединения кода программы с конкретными идентификаторами (ID).

Когда вы щелкаете кнопкой мыши по клавише Add Function, то появляется диалоговое окно, изображенное на рис. 9-33, для того чтобы подтвердить ваше имя функции.

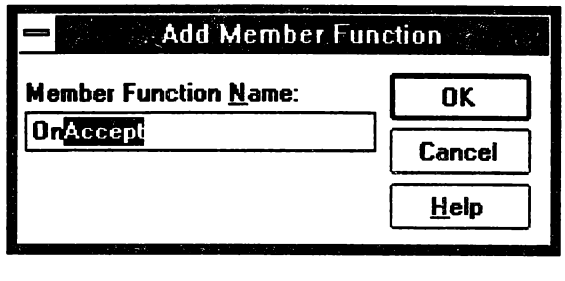


Рис. 9-33. Подтверждение вашего имени функции.

Имя файла по умолчанию, как правило, является соответствующим ситуации. После того как вы выбрали OK, появляется экран с программой, изображенный на рис. 9-34.

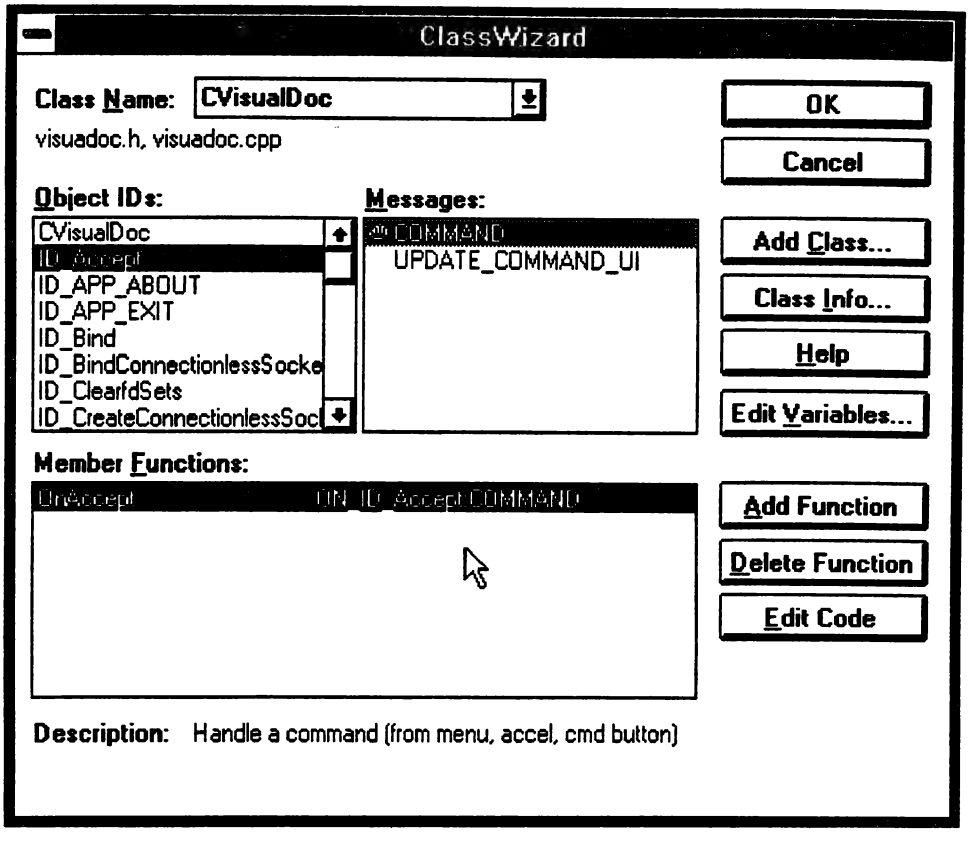


Рис. 9-34. Отображение добавленной компонентной функции.

Вы должны прикрепить все идентификаторы (ID), которые вы создали для кодирования фиктивных модулей программы (stubs). Вы можете упорядочить идентификаторы команд, так как они набраны на разных регистрах. Идентификаторы, набранные на верхнем регистре, являются резервными для Visual C++, и нет нужды прикреплять их в данном примере. Убедитесь в создании как функций COMMAND, так и версий UPDATE_COMMAND_UI каждого идентификатора. По окончании вы получите на экране картинку, изображенную на рис. 9-35.

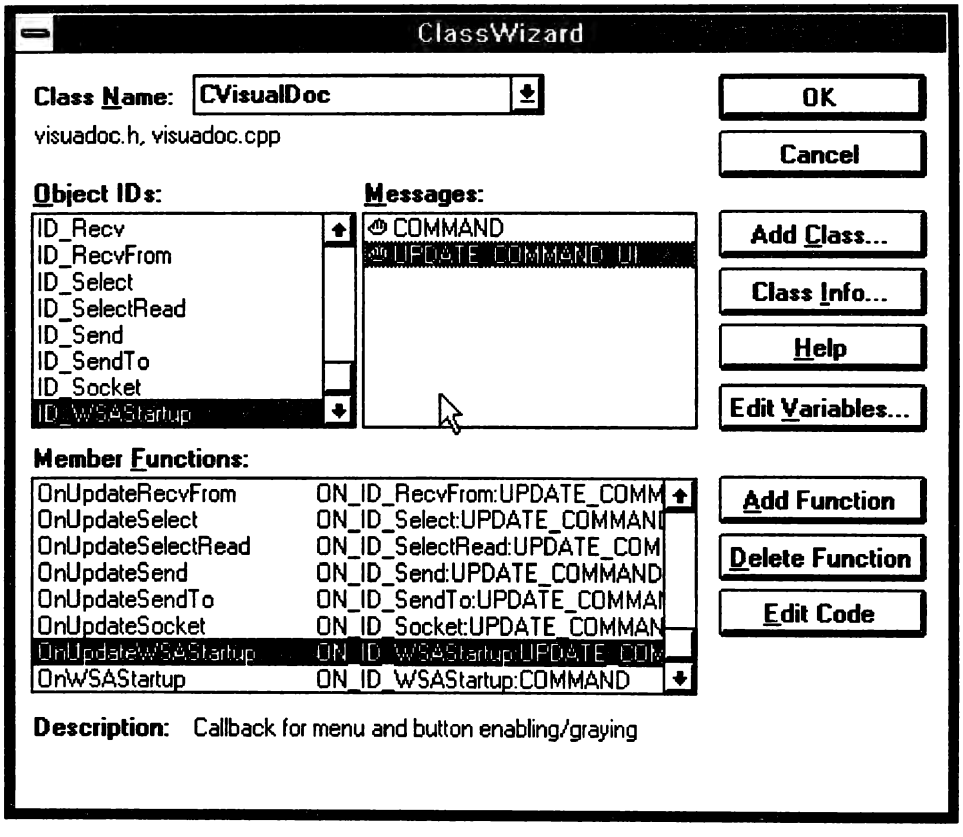


Рис. 9-35. Прикрепление командных идентификаторов (ID).

ОПИСАНИЕ КЛАССА CVISUALDOC

Класс CVisualDoc

Следующий код программы показывает класс CVisualDoc по умолчанию, который AppWizard строит в головном файле visuadoc.h:

```
// visuadoc.h : interface of the CVisualDoc class
//
/////////////////////////////////////////////////////////////////////////////
class CVisualDoc : public CDocument
 {
 protected: // create from serialization only
            CVisualDoc();
            DECLARE_DYNCREATE(CVisualDoc)
// Attributes
public:

// Operations
public:

// Implementation
public:
            virtual ~CVisualDoc();
            virtual void Serialize(CArchive& ar);   // overridden for document i/o
#ifdef _DEBUG
            virtual    void AssertValid() const;
            virtual   void Dump(CDumpContext& dc) const;
#endif
protected:
            virtual   BOOL     OnNewDocument();
// Generated message map functions
protected:
            //{{AFX_MSG(CVisualDoc)
            afx_msg void OnAccept();
            afx_msg void OnUpdateAccept(CCmdUI* pCmdUI);
            .
            .  Note: Code deleted for clarity; all that was here is simply
               declarations for all the Member Functions you just created



            afx_msg void OnUpdateWSAStartup(CCmdUI* pCmdUI);
            //}}AFX_MSG
            DECLARE_MESSAGE_MAP()
 };
/////////////////////////////////////////////////////////////////////////////
```

Целью класса документов является хранение особенностей режима регистрации и реагирование на выбор меню, который делает пользователь.

Далее приводится основной код программы, который должен присутствовать в головном файле для полного и точного описания класса CVisualDoc:

1. Определяются некоторые внешние глобальные переменные.

2. Заголовочная информация Userlib должна предшествовать самому классу. Данная заголовочная информация вызывается флагом USERLOG, который включает или отключает переназначение всех функций гнезда в классе.

3. Должен быть описан новый защищенный класс функций программы помощи. Эти функции используются внутри данного класса, и, так как они защищены, то только классы, являющиеся производными от класса CVisualDoc, будут иметь доступ к функциям программы помощи.

4. Определяются новые компоненты (members) данных для нового класса.

5. Описываются новые открытые компонентные функции (member functions).

VISUADOC.H (С ПОЯСНЕНИЯМИ)

```
// visuadoc.h : interface of the CVisualDoc class
//
/////////////////////////////////////////////////////////////////////////////

void output(int,char *);
extern int LogFlag;
extern char * Me;
extern char * NotMe;
```

Это место, где описаны внешние переменные. Данные переменные доступны в данном модуле, но определяются в других модулях этой программы. Выходная функция также описана, поэтому вы можете показать строки символов, не беспокоясь о непечатаемых символах.

```
#define USERLOG

#ifdef USERLOG
#define WSAStartup    UserLog_WSAStartup
#define socket        UserLog_socket
#define bind          UserLog_bind
#define listen        UserLog_listen
#define accept        UserLog_accept
#define connect       UserLog_connect
```

```
#define recv        UserLog_recv
#define send        UserLog_send
#define ioctlsocket  UserLog_ioctlsocket
#define select      UserLog_select
#endif

#include "portable.h"
```

Данный сегмент кода программы переопределяет функции гнезда для указания на функции гнезда User Logging (режим регистрации пользователя). Если вы хотите рекомпилировать программу без регистрации, просто удалите первую строку, которая определяет макрокоманду USERLOG. При этом #ifdef не выполняется и переназначения не произойдет.

```
Bclass CVisualDoc : public CDocument
{
protected: // create from serialization only
```

Защищенные функции могут быть вызваны только с помощью компонент класса или производных классов.

```
#ifdef USERLOG

int       UserLog_WSAStartup(WORD, WSADATA *);
SOCKET       UserLog_socket(int,int,int);
int       UserLog_bind(SOCKET,LPSOCKADDR,int);
void        UserShowAddress(LPSOCKADDR_IN);
int       UserLog_listen(SOCKET, int);
SOCKET       UserLog_accept(SOCKET,LPSOCKADDR,int *);
int       UserLog_connect(SOCKET,LPSOCKADDR,int);
int       UserLog_recv(SOCKET,char *,int,int);
int       UserLog_send(SOCKET,char *,int,int);
void        ErrorStop(void);
void        UserErrorMessage(char *);
void        UserGetData(SOCKET);
int       UserLog_ioctlsocket(SOCKET,long, unsigned long *);
long        UserLog_select(int,fd_set *,fd_set*,fd_set*, const struct timeval FAR *);
#endif
```

Это является описанием функций регистрации.

```
BOOL OnCmdMsg(UINT, int, void *, AFX_CMDHANDLERINFO *);
```

Данная функция используется для управления выбором меню. Так как мы произвели динамическое изменение меню fdSets, зависящее от фактических гнезд в определенном fd_set, необходимо ввести функцию, отслеживающую идентификаторы, связанные с данными пунктами меню. Visual C++ обеспечивает такую привязку в области обработки меню.

```
void output(int count, char *buffer);
```

Эта функция элемента обеспечивает такой же режим, как и регулярная выходная функция, однако, она позволяет осуществлять передачу данных в документ.

```
void UserExceptEach(fd_set *in,fd_set *all);
void UserReadEach(fd_set *in,fd_set *all);
void UserExceptList(fd_set *in,fd_set *all);
void UserReadList(fd_set *in,fd_set *all);
```

Эти функции обрабатывают все гнезда в fd_set, которые указаны. Вводится также весь набор fd_set, так что в случае возникновения ошибки во время обработки гнезда она может быть удалена.

```
void ActivateSocket(UINT nId);
```

Данная функция активизирует определенное гнездо. Следует отметить, что меню ID для этого гнезда установлено. Это тот ID, который был присвоен данному гнезду динамически, путем подбора меню подсистемы. Ввиду того, что наше меню fd_sets является динамическим, вы должны создать схему меню ID.

```
CVisualDoc();
DECLARE_DYNCREATE(CVisualDoc)

// Attributes
public:
  CStringList                m_LineList;
```

Класс документов должен обеспечивать место для хранения информации, содержащейся в документе. Так как регистрируемая информация находится в данном документе, то самым приемлемым местом для ее хранения является список строк символов. Базовые классы фирмы Microsoft (Microsoft Foundation Classes — MFC) обеспечивают предварительно определенный класс *CStringList*, который является реализацией списка строк. Переменная **m_LineList** описана здесь как *CStringList*.

```
CSize                      m_sizeDoc;
```

Предполагается также, что документ знает, какого он размера. MFC под названием CSize является способом, с помощью которого документ предоставляет эту информацию. Здесь объявляется переменная **m_sizeDoc** для обеспечения места для хранения размера документа.

```
// Operations
public:
```

Эти открытые функции представлены в таком виде, что вы можете обратиться к ним для осуществления операций. Открытые функции можно вызывать извне компонентных функцией класса.

```
int                 dprintf(char *,...);
```

Данная функция обеспечивает возможности, аналогичные *printf*, но передает информацию в документ.

```
void                DeleteContents();
```

Данная функция удаляет всю информацию в документе.

```
POSITION            GetFirstStringPos();
```

Данная функция возвращает структуру POSITION, которая указывает на то место, где в документе находится первая строка. Поэтому для осуществления поиска строки вы можете использовать функцию *GetNextString*.

```
CString&            GetNextString(POSITION &);
```

Данная функция осуществляет поиск строки, расположенной на указанной позиции (POSITION). Переменная POSITION обновляется для указания на следующую строку в списке.

```
CSize               GetDocSize() {return m_sizeDoc;}
```

Данная функция возвращает размер документа.

```
HMENU               MakeMenu(fd_set &,char,UINT);
```

Данная функция создает MENU из fd_set, которое представляется в виде параметра.

```
int                 UserReadClient(SOCKET);
```

Эта функция считывает данные из определенного гнезда.

```
int                 UserSendClient(SOCKET);
```

Данная функция передает данные на определенное гнездо.

```
// Implementation
public:
    virtual ~CVisualDoc();
    virtual void Serialize(CArchive& ar);    // overridden for document i/o
#ifdef _DEBUG
```

```
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif
protected:
        virtual     BOOL     OnNewDocument();

// Generated message map functions
protected:
        //{{AFX_MSG(CVisualDoc)
        afx_msg void OnAccept();
        afx_msg void OnUpdateAccept(CCmdUI* pCmdUI);
        afx_msg void OnBind();
        afx_msg void OnUpdateBind(CCmdUI* pCmdUI);
        .
        .
        .
        afx_msg void OnSocket();
        afx_msg void OnUpdateSocket(CCmdUI* pCmdUI);
        afx_msg void OnWSAStartup();
        afx_msg void OnUpdateWSAStartup(CCmdUI* pCmdUI);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
```

РЕАЛИЗАЦИЯ КЛАССА CVISUALDOC

Приведенный ниже список указывает части кода программы, которые предназначены для вашего использования:

- Должны быть определены глобальные переменные.
- Должны быть определены функции глобального помощника.
- Должен быть реализован конструктор.
- Должны быть реализованы функции обработки документа:
  — *DeleteContents*,
  — *GetFirstStringPosition*,
  — *GetNextString*,
  — *GetNextLine* (*функция помощника*),
  — *dprintf*,
  — *output*.
- Реализуются функции помощника.
- Должны быть реализованы все команды Меню.

НОВЫЙ VISUADOC.CPP (С ПОЯСНЕНИЯМИ)

```
// visuadoc.cpp : implementation of the CVisualDoc class
//

#include "stdafx.h"
#include "visual.h"
#include "senddial.h"    // added to define Send-Dialog
#include "portdial.h"    // added to define Port-Dialog

#include "visuadoc.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// Global Variables
  WORD        VersionRequested;        // Version for WSAStartup
  WSADATA     ImplementationDetails;   // Return Value form WSAStartup
  SOCKET      ServerSocket;            // First Socket
  SOCKET      ClientSocket;            // Socket id of a client
  SOCKADDR_IN   ServerSocketAddress;   // Address info for Server Socket
  SOCKADDR_IN   ClientSocketAddress;   // Address info for Client Socket
  fd_set      AllSockets;              // List of all active sockets
  fd_set      ReadSockets;             // List of all sockets for reading
  fd_set      WriteSockets;            // List of all sockets for writing
  fd_set      ExceptSockets;           // List of all sockets with OOB data
  int         rv;                      // Temporary Return Value
  long        lrv;                     // LONG Return Value
  int         LogFlag=1;               // Detailed Logging Enabled?
  int         WaitFlag =0;             // Not used in this example
  int         ReadFlag=0;              // OOB data enabled?
  char *      Me="Server";             // I am the server
  char *      NotMe="Client";          // They are the Clients
```

Эти глобальные переменные являются родовыми переменными, с которыми мы уже встречались в других примерах программирования. Данные переменные используются для хранения различных адресов и гнезд, а также для записи различных состояний флагов.

```
// UI Flags  0 = No Check, 1 = Checked, 2 = Dot

  int     WSAStartupCheck = 1;
  int     SocketCheck=0;
  int     BindCheck=0;
  int     ListenCheck =0;
  int     InsertCheck =0;
```

```
int     InitializeCheck=0;
int     SelectCheck =0;
int     InstallBHFlag = 0;
```

Данные переменные используются функциями Update Command UI для определения того, какие пункты меню должны быть помечены галочкой, точкой или не помечены ничем. Это дает подсказку конечному пользователю и помогает ему выбрать правильный пункт меню.

```
// Application external variable

extern CServerApp theApp;          // defined in visual.cpp
```

Данная переменная используется для обращения к отдельным частям прикладной программы. Эта переменная определена в файле *visual.cpp* и описана в данном файле как внешняя.

```
/////////////////////////////////////////////////////////////////////////////
// Global Helper Functions

// Copy all the sockets located in the in fd_set to the out fd_set
static void UserFd_copy(fd_set *out, fd_set *in)
 {
 unsigned int i;

  for (i=0;i<in->fd_count;i++) out->fd_array[i]=in->fd_array[i];
 out->fd_count = in->fd_count;
```

Данная функция является родовой и используется для копирования одного fd_set в другой.

```
// add one to each letter in the string pass in as temp void
UserWork(char *temp)
 {
  for(;' \0'!=*temp;temp++) (*temp)++;
 }
```

Это родовая функция, предназначенная для добавления единицы к каждой букве строки символов.

```
/////////////////////////////////////////////////////////////////////////////
// CVisualDoc

IMPLEMENT_DYNCREATE(CVisualDoc, CDocument)

BEGIN_MESSAGE_MAP(CVisualDoc, CDocument)
 //{{AFX_MSG_MAP(CVisualDoc)
```

```
            ON_COMMAND(ID_Accept, OnAccept)
            ON_UPDATE_COMMAND_UI(ID_Accept, OnUpdateAccept)
            ON_COMMAND(ID_Bind, OnBind)
            ON_UPDATE_COMMAND_UI(ID_Bind, OnUpdateBind)
            ON_COMMAND(ID_BindConnectionlessSocket,
            OnBindConnectionlessSocket)
            .
            .
            .
            ON_UPDATE_COMMAND_UI(ID_BindConnectionlessSocket,
            OnUpdateBindConnectionlessSocket)
            ON_COMMAND(ID_Socket, OnSocket)
            ON_UPDATE_COMMAND_UI(ID_Socket, OnUpdateSocket)
            ON_COMMAND(ID_WSAStartup, OnWSAStartup)
            ON_UPDATE_COMMAND_UI(ID_WSAStartup, OnUpdateWSAStartup)
            //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CVisualDoc construction/destruction

// Construct a CVisualDoc Class
// m_LineList is a collection class which contains a list of lines
// an empty string is created and added to the list when the document is
// first created. The size of the document is set to be 1 character wide by
// 1 line long.
 CVisualDoc::CVisualDoc()
{
            CString    temp;
            m_LineList.AddTail(temp);
            m_sizeDoc = CSize(1,1);
}
```

После создания исходного документа инициализируются его два компонента данных.

Переменная m_LineList имеет исходную пустую строку, а m_sizeDoc устанавливается равным одному знаку по длине на один знак по ширине.

```
CVisualDoc::~CVisualDoc()
{
}
```

Возможно использование деструктора по умолчанию.

```
BOOL CVisualDoc::OnNewDocument()
{
      if (!CDocument::OnNewDocument())
         return FALSE;
     // TODO: add reinitialization code here
     // (SDI documents will reuse this document)
     return TRUE;
}
```

Ввиду того что вы ничего не храните и у вас больше нет пункта меню New, вы не должны использовать данную функцию или функцию *Serialize*, приведенную ниже.

```
/////////////////////////////////////////////////////
// CVisualDoc serialization

void CVisualDoc::Serialize(CArchive& ar)
 {
   if (ar.IsStoring())
   {
     // TODO: add storing code here
    }
    else
    {
      // TODO: add loading code here
     }
 }
/////////////////////////////////////////////////////
// CVisualDoc diagnostics

#ifdef _DEBUG
void CVisualDoc::AssertValid() const
{
                    CDocument::AssertValid();
}

void CVisualDoc::Dump(CDumpContext& dc) const
{
                    CDocument::Dump(dc);
}

#endif //_DEBUG
```

Функции диагностики приемлемы без каких-либо изменений.

```
//////////////////////////////////////////////////////////////////////
// CVisualDoc Document manipulation functions

void CVisualDoc::DeleteContents()
  { while(m_LineList.IsEmpty())
          {CString Temp = m_LineList.RemoveHead();
 }
 }
```

Данная функция циклически пробегает по всем строкам, хранящимся в данное время в *CStringList m_LineList*, перемещает данные и выбрасывает их.

```
POSITION CVisualDoc::GetFirstStringPos()
  {return (m_LineList.GetHeadPosition());
  }
```

CStringlist m_LineList содержит компонентную функцию, к которой вы обращаетесь для выхода в первую строку, хранящуюся в списке. Это величина, которую вы возвращаете.

```
CString& CVisualDoc::GetNextString(POSITION &rPos)
  { return(m_LineList.GetNext(rPos));
  }
```

CStringList m_LineList содержит другую компонентную функцию, к которой вы обращаетесь для отыскания 1CString 0, находящейся на определенной позиции, и, в качестве побочного эффекта, она обновляет переменную позиционирования.

```
static BOOL GetNextLine(CString &last, CString &main)
  { int RetLocation; // Return (\n) location

    if (0==main.GetLength()) return(0);  // if main empty, return false

        if (-1==last.Find('\n'))  // if \n not found in last do this block
          { RetLocation = main.Find('\n');
            if (-1 == RetLocation)
                  { last += main;  // concatenate main to last
                    main = "";    // zap main to be empty
                  }
                else                // otherwise, concatenate only up to \n
                  {last += main.Left(RetLocation+1);
                   main = main.Right(main.GetLength() - RetLocation -1);
                  }
          }
        return(1);
  }
```

Это функция помощника. Следует отметить, что это не реальный элемент класса VisualDoc. Цель данной функции — позволить функциям дисплея свободно распечатывать данные регистрации, хранимые в *CStringList m_LineList*. Вы хотите, чтобы функция *dprintf* работала таким же образом, как при печати в *stdout*; поэтому, если строка не оканчивается на \n, то следующая строка в *LineList* должна появиться в следующей символьной позиции, а не в следующей строке. Данная функция имеет следующую логику:

Конкатенируйте следующую строку с окончанием \n из *main* в *last*; если *last* имеет окончание \n, просто возвратитесь.

Даны два массива — *last* и *main*. Массив *main* включает в себя многострочный массив, потенциально содержащий много новых строк с \n, или, возможно, не содержащий их вообще. Массив *last* включает в себя строку, которая 1 0может заканчиваться, а может и не заканчиваться символами \n. Выполните следующее:

Если *main* пуста, возвратите значение false ("ложно"). — В *main* нет следующей строки.

Если *last* оканчивается на \n, возвратите значение true ("истинно"). — В *last* имеется строка, оканчивающаяся на \n.

Если *last* не оканчивается на \n, тогда переместите (MOVE) все, включая \n, из *main* в *last* и возвратите значение "истинно".

Заметьте, что *main* теперь меньше на одну строку, а *last* содержит больше данных.

Теперь для *main* существуют две возможности:

1. Она пуста. На вызов данной функции в следующий раз последует отказ.

2. Она содержит больше строк данных.

```
int CVisualDoc::dprintf(char *input, ...)
 {
 int rv;
 char buffer[5000];
 int LineLength=m_sizeDoc.cx;

 va_list va_alist;
 va_start(va_alist,input);
 rv = vsprintf(buffer,input,va_alist);

 CString last= m_LineList.RemoveTail();
 CString temp(buffer);

 while(GetNextLine(last,temp))
  {
      m_LineList.AddTail(last);
      if (last.GetLength() > LineLength) LineLength = last.GetLength();
      last = "";
     }
 m_sizeDoc.cx=LineLength;
 m_sizeDoc.cy=m_LineList.GetCount()+1;
 UpdateAllViews(NULL);
 return(rv);
}
```

Это функция, которая на деле реализует *printf*. Для операций с аргументами переменной используется стандартный алгоритм С. Для печатания данных в локальный буфер строки объемом 5000 байт используется *vsprintf*. Для получения последней строки в *LineList* обратитесь к компонентной функции *GetTail* из класса *CStringList*. После этого используйте функцию *GetNextLine* следующим образом:

- Если предыдущая строка не оканчивается на \n, то все, что было в *temp*, переносится в *last*, а *last* присоединяется к концу *LineList*. В следующий раз при обращении к *GetNextLine* цикл *while* завершается.

Результаты: текущая строка присоединяется к концу существующей (не оканчивающейся на \n) строки.

- Если предыдущая строка оканчивается на \n, то *GetNextLine* возвращается без выполнения каких-либо действий; поэтому строка, которую вы только что удалили из *LineList*, возвращается обратно. В следующий раз при вызове *GetNextLine* текущая строка возвращается с тем, чтобы быть затем присоединенной к концу *LineList*. В следующий раз, при вызове *GetNextLine* она завершает цикл *while*.

Результаты: предыдущая строка снова возвращена в список; после этого текущая строка добавляется к списку.

```
void CVisualDoc::output(int count, char *buffer)
 {
 while (count > 0)
  {
   if (31>*buffer) dprintf("\\%03d",*buffer);
   else      dprintf("%c",*buffer);

   buffer++;
   count—;
   }
 dprintf("\n");
 }
```

Это является реализацией компонентной функции вывода.

```
int CVisualDoc::UserSendClient(SOCKET ClientSocket)
 {
 static int i; // Used to keep track of the message number
 static int  ooi;  // Used to keep track of the OOB message #
```

Данные статические переменные будут помнить свои значения с момента их предыдущего вызова.

```
char *  OOMessage;
char *   data;
int  k;
```

Данные переменные используются для хранения данных, которые будут изменены в зависимости от того, имеете ли вы дело с внедиапазонным (OOB Out-Of-Band) или обычным сообщением.

```
char   buffer[80];                  // Temporary buffer for sprintf

if (1==ReadFlag)                    // this is an oob Send
  {   OOMessage = "OOB ";
      k = ooi;
  }
else                                // this is a normal send
  { OOMessage = "";
      k = i;
  }
sprintf(buffer,"%sMessage %d from Server\n",OOMessage,k);
```

Теперь в любом случае буфер содержит правильное сообщение:

```
["]Message [i] from Server
```

или:

```
["OOB "]Message [ooi] from Server.
```

```
SendDialog     output;
output.m_String = buffer;
if (IDCANCEL == output.DoModal()) {return(0);}
data = output.m_String.GetBuffer(output.m_String.GetLength());
```

Данный сегмент кода программы описывает диалоговое окно class[SendDialog], инициализирует компоненту данных *m_String*, а затем отображает диалоговое окно на дисплее. Если пользователь отменяет диалоговое окно, вы возвращаетесь без осуществления дальнейшей обработки. В противном случае, вы копируете данные из CString и устанавливаете указатель символов в строку. *GetBuffer* и *GetLength* являются компонентными функциями класса CString. Теперь данные переменной указывают на строку символов, которую пользователь набрал в диалоговом окне.

```
if (1==ReadFlag) ooi++;
else      i++;
```

Увеличьте соответствующую статическую переменную таким образом, чтобы при обращении к этой функции в следующий раз сообщение по умолчанию получило бы следующий порядковый номер.

```
rv = send (ClientSocket,data,strlen(data),ReadFlag);
```

Вызовите функцию *send* для передачи данных в сеть 1.

```
if (SOCKET_ERROR == rv)
      switch(WSAGetLastError())
       { case(WSAECONNRESET):
         dprintf("(%d) Connection Reset\n",ClientSocket);
         return(0);    // signals error
        default: ErrorStop();
               return(0);
       }
               else return(1);    // signals success
 }
```

Проверьте сетевые ошибки и возвратите соответствующее значение вызывающей программе для указания на то, произошла ошибка или нет.

СОЗДАНИЕ SENDDIALOG

Введение

Предыдущая функция содержала обращение к диалоговому окну под названием SendDialog. Для создания класса, обеспечивающего инкапсуляцию (скрытие) данного диалога, необходимо использовать App Studio. Активизируйте App Studio путем выбора пункта меню из Tools Menu.

Выберите опцию New из меню Resource для получения диалогового окна новых ресурсов:

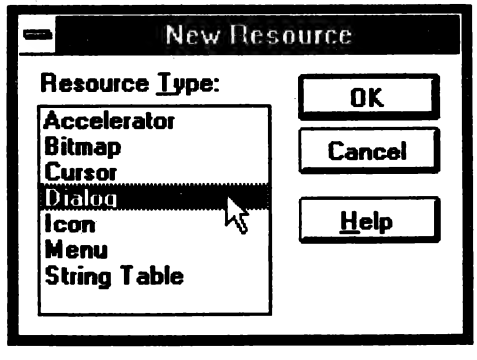


Рис. 9-36. Диалоговое окно New Resource.

После выбора создания нового диалога появляется Dialog Editor (Редактор диалога) (см. рис. 9-37).

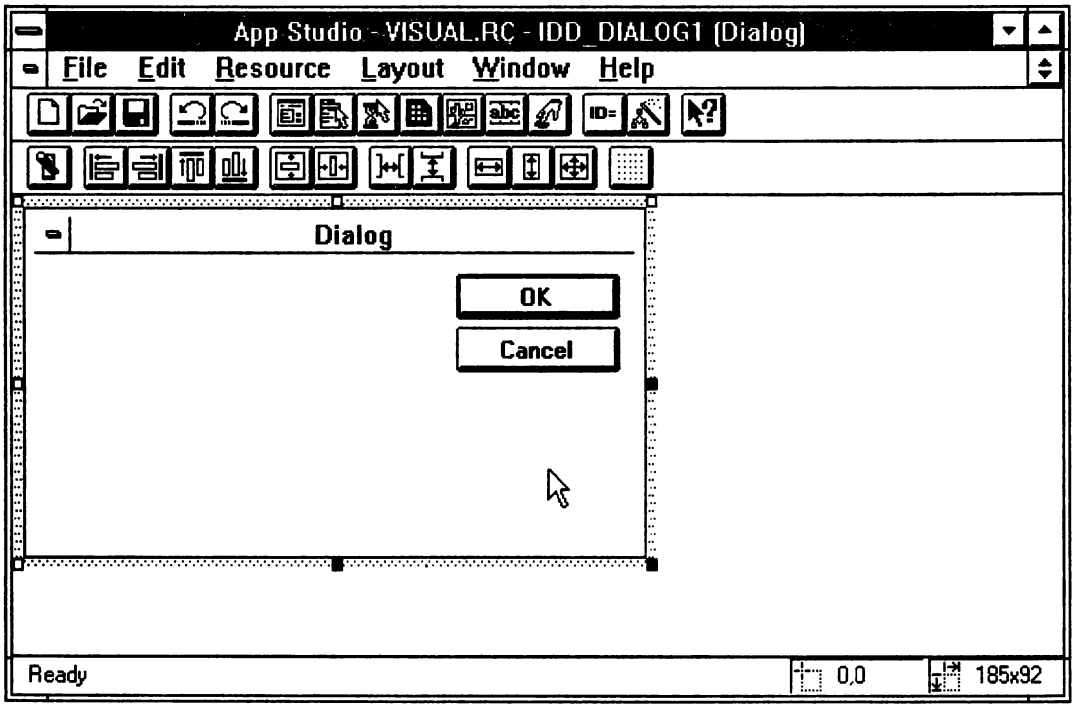


Рис. 9-37. Редактор диалога.

При осуществлении двойного щелчка кнопкой мыши внутри диалогового окна окно Dialog Properties появится в виде, показанном на рис. 9-38.

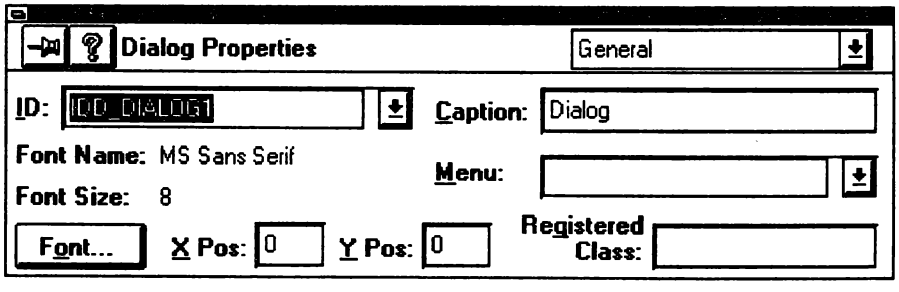


Рис. 9-38. Диалоговое окно Dialog Properties.

Измените ID с IDD_DIALOG1 на IDD_SendDialog и измените заголовок с Dialog на Send a Message.

Закройте диалоговое окно Properties, щелкнув мышью по клавише Close в дальнем левом верхнем углу экрана, а затем для показа на экране панели управления палитрой (см. рис 9-39), нажмите клавишу <F2>.

Рис. 9-39. Панель управления палитрой.

Вы хотите создать два типа управления. Первый тип — поле для пользователей, чтобы выводить в нем данные, а второй тип — текст для маркировки поля. Щелкните мышью по клавише Edit Control для создания нового средства управления редактированием. Щелкните мышью по клавише Static Text для создания статического текста. Перегруппируйте диалоговое окно так, чтобы оно выглядело, как показано на рис. 9-40.

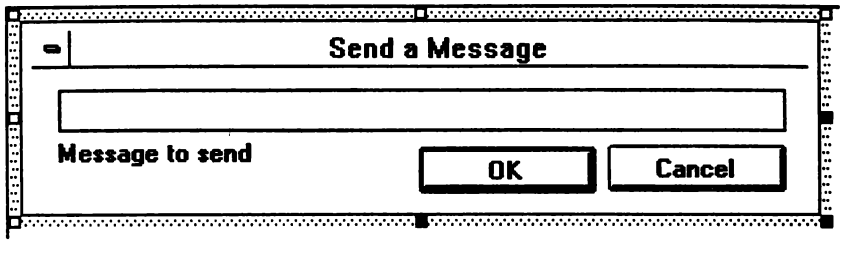


Рис. 9-40. Перегруппированное диалоговое окно.

После того как вы создали окно диалога, используя редактор диалогового окна, создайте класс, который будет инкапсулировать диалоговое окно. Выберите пункт ClassWizard из меню Resource, входящего в App Studio. При этом должна появиться подсказка о введении имени класса, который вы создаете. Введите SendDialog:

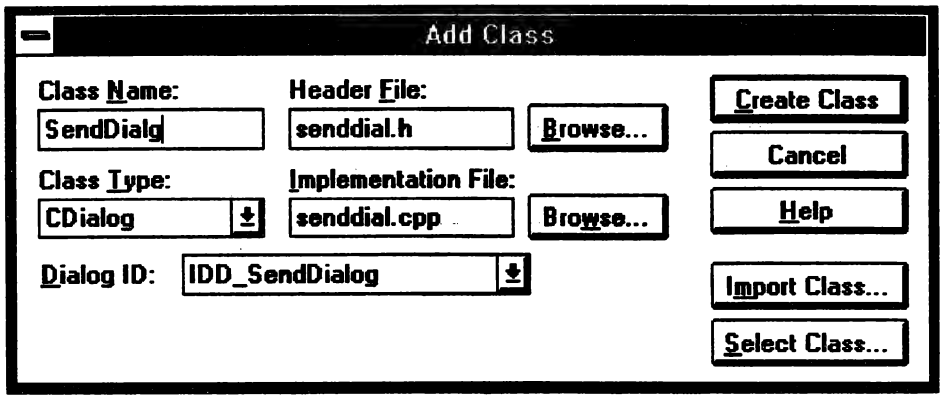


Рис. 9-41. Создание класса.

Щелкните мышью по клавише Create Class для получения на экране окна ClassWizard. Далее выберите кнопку Edit Variables для получения приведенной ниже картинки:

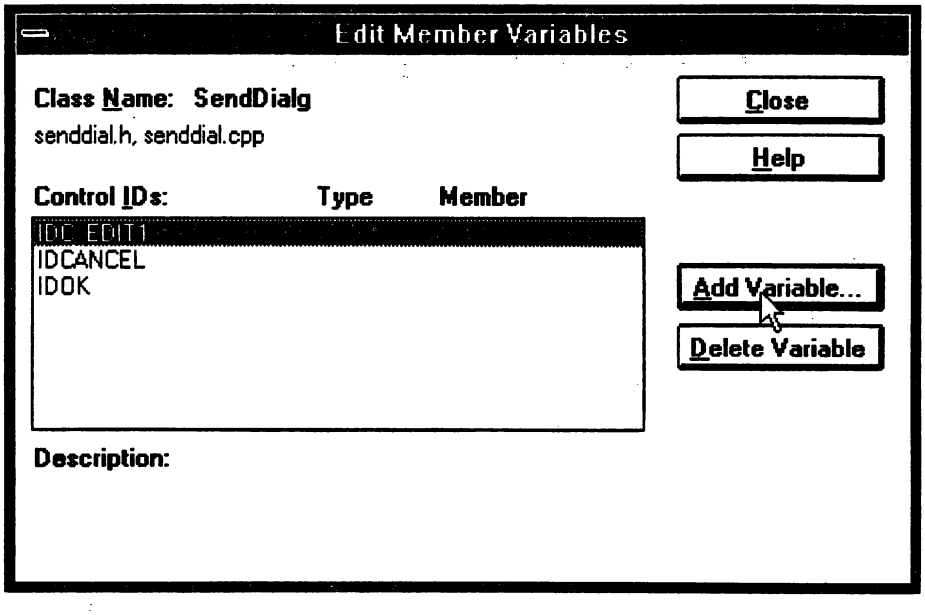


Рис. 9-42. Окно редактора переменных компонент.

Здесь вы должны обеспечить сочетание компоненты данных в создаваемом вами классе с управлением в окне диалога, которое вы создали ранее. Единственное, что может вас беспокоить в этом, — это содержание управления edit. Оно содержит значение ID из ID_EDITI. Выберите этот ID в окне списка, а затем щелкните мышью по клавише Add Variable.

Вы получите на экране картинку, показанную на рис. 9-43.

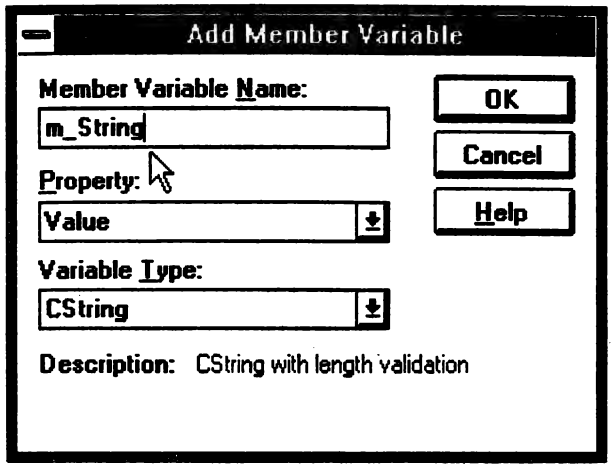


Рис. 9-43. Меню Add Variable (Добавление переменной).

Вам необходимо будет ввести имя создаваемой переменной. Выберите OK, закройте ClassWizard и выйдите из App Studio. Сохраните все сделанные вами изменения. Теперь SendDialog Class разработан и создан для вас автоматически. Вы можете создать переменную SendDialog и обратиться к компонентам SendDialog следующим образом:

```
SendDialog X;

X.m_String = "Hello";

X.doModal();

printf("Data %s\n",(char *) X.m_String);
```

Этот код программы создает экземпляр переменной, называемой X, присваивает компоненте *m_String* значение "Hello", вызывает функцию DoModal для реализации окна диалога, а затем распечатывает данные пользователя в поле Edit. Преобразование (char*) в операторе *printf* должно превратить значение *m_String* переменной CString в указатель *char**, подходящий для *printf*. Для вас это реализовано в классе CString.

VISUADOC.CPP (С ПОЯСНЕНИЯМИ)

```
int CVisualDoc::UserReadClient(SOCKET ClientSocket)
 {
   char    buffer[100];
   rv = recv (ClientSocket,buffer,80,ReadFlag);
   if (SOCKET_ERROR == rv)
     switch(WSAGetLastError())
      { case(WSAECONNRESET):
       dprintf("(%d) Connection Reset\n",ClientSocket);
       return(0);    // signals error
       default: ErrorStop();
      }
   if (0==rv)
      {dprintf("Zero Length Data interpreted as a network disconnect\n");
       dprintf("(%d)Connection Reset\n");
       return(0);    // signals error
      }

   dprintf("socket:%03d>[%03d]",ClientSocket,rv);
   output(rv,buffer);   // output is used because buffer may contain NUL characters
   if (0==strcmp("exit",buffer))
    { dprintf("(%d)Client sent Exit commandt\n",ClientSocket);
      return(0);    // signals error
    }
   if (0==strcmp("quit",buffer))
    { dprintf("(%d)Client sent quit command\n",ClientSocket);
      return(-1);    // signals to exit the server
    }

   if (0==strcmp("slog",buffer))
    { LogFlag = (1==LogFlag) ? 0: 1;
      dprintf("Client sent slog command. LogFlag now %d\n",LogFlag);
      return(1);    // signals normal      }

   if (0==strcmp("swait",buffer))
    { WaitFlag = (1==WaitFlag) ? 0: 1;
      dprintf("Client sent swait command. WaitFlag now %d\n",WaitFlag);
      return(1);  // signals normal
    }
   return(1);    // signals normal
 }
```

Данная функция вызывает функцию *recv* для обеспечения считывания данных с идентифицированного сетевого гнезда. По существу, это та же самая функция, которая использовалась в других примерах.

```
void CVisualDoc::UserExceptEach(fd_set *in,fd_set *all)
 {
  unsigned int i;
  ReadFlag = MSG_OOB;
  for (i=0;i<in->fd_count;i++)
   {
    printf("**** (%d) Out of Band Data\n",in->fd_array[i]);
    UserReadClient(in->fd_array[i]);
    printf("**** End of this Out of Band packet\n");
   }
  ReadFlag = 0;
 }

void CVisualDoc::UserReadEach(fd_set *in,fd_set *all)
 {
 unsigned int i;
 ReadFlag = 0;
 for (i=0;i<in->fd_count;i++)
  {
  if (ServerSocket==in->fd_array[i]) continue;    // Ignore the ServerSocket
  rv = UserReadClient(in->fd_array[i]);
  if (0==rv)
   {
    dprintf("(%d) Socket Error - Socket being closed\n",in->fd_array[i]);
    FD_CLR(in->fd_array[i],all);    // Remove the Sockets from fd_set all
   }
  }
 }
```

Эти две функции используются для проведения цикла по всем гнездам в соответствующем fd_set. По существу, это те же функции, что и в предыдущих примерах.

```
void CVisualDoc::UserExceptList(fd_set *in,fd_set *all)
 {
 unsigned int i;
 for (i=0;i<in->fd_count;i++)
  {
  if (ServerSocket!=in->fd_array[i])
   dprintf("**** (%d) Out of Band Data on Socket\n",in->fd_array[i]);
  }
 }

void CVisualDoc::UserReadList(fd_set *in,fd_set *all)
 {
 unsigned int i;
 for (i=0;i<in->fd_count;i++)
  {
    if (ServerSocket==in->fd_array[i]) continue;
```

```
      dprintf("Data Ready on Socket (%d)\n",in->fd_array[i]);
    }
  }
```

Эти две функции используются для составления списка гнезд, которые в настоящий момент содержат информацию. Данные функции только распечатывают информацию; они не обеспечивают обработку гнезд. Для обработки гнезд следует использовать *UserExceptEach* и *UserReadEach*.

```
/////////////////////////////////////////////////////////////////////////
// CVisualDoc Socket Functions Menu Commands

void CVisualDoc::OnWSAStartup()
 {
  int rv;
  if (2==WSAStartupCheck)
     dprintf("Warning:Already called WSAStartup, should only call it once\n");

  VersionRequested = MAKEWORD(1,1);
  rv = WSAStartup (VersionRequested, &ImplementationDetails);
  if (0 != rv ) ErrorStop();

  WSAStartupCheck=2;
  SocketCheck=1;
}
```

Данная функция инициализирует сеть, затем задает переменные UI путем выставления точки (2) перед пунктом меню WSAStartup и галочки (1) перед пунктом меню 1st Socket. Если функция вызывается дважды, то на печать выводится предупреждающее сообщение, хотя сама функция выполняется дважды.

```
void CVisualDoc::OnSocket()
 {
  if (2==SocketCheck)
     dprintf("Warning:Already called socket, should only call it once\n");

  ServerSocket = socket (PF_INET,SOCK_STREAM,0);
  if (INVALID_SOCKET == ServerSocket) ErrorStop();

  SocketCheck=2;
  BindCheck = 1;
}
```

Данная функция создает главное гнездо путем обращения к функции *socket*, а затем задает переменные UI путем выставления точки (2) перед пунктом меню 1st Socket и галочки (1) перед пунктом меню Bind. Если функция вызывается дважды, то на печать выводится предупреждающее сообщение, хотя сама функция выполняется дважды. В случае возникновения ошибки при создании главного гнезда распечатывается сообщение об ошибке.

```
void CVisualDoc::OnBind()
 { PortDialog input;

   if (2==BindCheck)
      dprintf("Warning:Already called bind, should only call it once\n");

   input.m_port = 100;  // default port
   if (IDCANCEL == input.DoModal()) return;

   ServerSocketAddress.sin_family            = AF_INET;      // InterNet
   ServerSocketAddress.sin_addr.S_un.S_addr    = INADDR_ANY;   // Any Address
   ServerSocketAddress.sin_port              = input.m_port;

   rv = bind (ServerSocket,(LPSOCKADDR)&ServerSocketAddress,sizeof(ServerSocketAddress));
   if (0 != rv ) ErrorStop();

   BindCheck=2;
   ListenCheck=1;
 }
```

Данная функция подсказывает пользователю ввести номер порта, а затем связывает сервер с сетью через этот порт. По умолчанию номер порта 100. После того как функция связала сервер, UI обновляется, чтобы показать, что следующий логический шаг — Listen.

```
void CVisualDoc::OnListen()
 {
  if (2==ListenCheck)
   dprintf("Warning:Already Called Listen, should only call it once\n");

  rv = listen (ServerSocket,5);
  if (0!= rv) ErrorStop();

  ListenCheck=2;
  InsertCheck=1;
}
```

Данная функция вызывается для трансформации гнезда main в гнездо listen. Устанавливается очередь из пяти клиентов. Переменная UI (пользовательского интерфейса) обновляется для того, чтобы показать, что следующий логический шаг — пункт Insert.

```
void CVisualDoc::OnInsert()
 {
  static int SocketNumber=0;
  if (2==InsertCheck)
     dprintf("Warning:Already called insert, should only call it once\n");

  FD_SET(ServerSocket,&AllSockets);
```

```
  dprintf("Inserted the Listen Socket (%d) into the fdSet [All]\n",ServerSocket);
  dprintf(" Look in the fdSets menu to verify\n");
  InsertCheck =2;
  InitializeCheck=1;
}
```

Данная функция используется для вставки гнезда Listen в *ALL Sockets* из fdSet. Переменная UI задается таким образом, что следующая логическая функция должна инициализировать fdSets.

```
void CVisualDoc::OnInitialize()
{
   UserFd_copy(&ReadSockets,&AllSockets);
   UserFd_copy(&ExceptSockets,&AllSockets);
   UserFd_copy(&WriteSockets,&AllSockets);
   dprintf("Copied the fdSet [All] to the other fdSets [Look at FdSet Menu to Verify]\n");
   InitializeCheck = 2;
   SelectCheck=1;
}
```

Данная функция инициализирует все fd_sets путем копирования *All Sockets* в другие fd_sets. Для выполнения копирования используется функция помощника. После этого UI обновляется для того, чтобы показать, что затем должна быть активирована функция Select.

```
void CVisualDoc::OnSelect()
{
   Irv = select (0,&ReadSockets,&WriteSockets,&ExceptSockets,NULL);
   if (SOCKET_ERROR == Irv) ErrorStop();
   InitializeCheck=1;
   SelectCheck=0;
}
```

Данная функция используется для вызова функции *select*. Следует заметить, что она переходит во все fd_sets, как только вы выполняете операцию Select [R+W]. Переменная UI обновляется для указания на то, что следующим логическим шагом будет ее запуск.

```
void CVisualDoc::OnSelectRead()
{
   FD_ZERO(&WriteSockets);
   Irv = select (0,&ReadSockets,NULL,&ExceptSockets,NULL);
   if (SOCKET_ERROR == Irv) ErrorStop();
   InitializeCheck=1;
   SelectCheck=0;
}
```

Данная функция аналогична предыдущей, однако, fd_set Write передается как NULL, поэтому она не будет управлять какой-либо операцией записи. Как и ранее, UI продолжает обновляться.

```
void CVisualDoc::OnClearfdSets()
 {
     FD_ZERO(&AllSockets);
     FD_ZERO(&ReadSockets);
     FD_ZERO(&WriteSockets);
     FD_ZERO(&ExceptSockets);
     dprintf("All fdsets have been cleared\n");
}
```

Данная функция позволяет вам очистить все fd_sets. Вызов данной функции выходит за рамки обычной операции, однако, ее вызов может быть использован в качестве пункта меню, когда вы хотите очистить все ваши гнезда.

```
/////////////////////////////////////////////////////////////////////////
// CVisualDoc Actions Menu Commands

void CVisualDoc::OnAccept()
 {
   if (FD_ISSET(ServerSocket,&ReadSockets))
    {
     ClientSocket = accept (ServerSocket,(LPSOCKADDR) &ClientSocketAddress,NULL);
     if (INVALID_SOCKET == ClientSocket) ErrorStop();

     dprintf("A Client has been attached(%d)\n",ClientSocket);
        dprintf(" Verify this by looking in the fdSets Menu\n");
     FD_SET(ClientSocket,&AllSockets);
     FD_CLR(ServerSocket,&ReadSockets);
    }
 }
```

Данная функция вызывается для принятия нового клиента. После этого новый клиент вставляется в набор All Sockets fd_set.

```
void CVisualDoc::OnRecv()
 {
  UserReadEach(&ReadSockets,&AllSockets);
  FD_ZERO(&ReadSockets);
 }
```

Данная функция будет пытаться получить *(recv)* данные со всех гнезд в *ReadSockets* fd_set. После получения данных fd_set очищается, так что больше уже не будет появляться в меню.

```
void CVisualDoc::OnSend()
 {
  unsigned int j;
  ReadFlag = 0;
  for (j=0;j<WriteSockets.fd_count;j++)
   {
```

```
    UserSendClient(WriteSockets.fd_array[j]);
   }
  FD_ZERO(&WriteSockets);
 }
```

Данная функция будет пытаться отправить данные на все гнезда в наборе WriteSockets fd_set. После отправки данных fd_set очищается так, что больше уже не будет появляться в меню.

```
void CVisualDoc::OnOOBRecv()
 {
  UserExceptEach(&ExceptSockets,&AllSockets);
  FD_ZERO(&ExceptSockets);}
 }
```

Данная функция будет пытаться получить исключительные данные из всех гнезд в наборе *ExceptSockets* fd_set. После получения данных fd_set очищается так, что больше уже не будет появляться в меню.

```
void CVisualDoc::OnOOBSend()
 {
  unsigned int j;
  ReadFlag = MSG_OOB;
  for (j=0;j<WriteSockets.fd_count;j++)
   {
    printf("**** (%d) Out of Band Data\n",WriteSockets.fd_array[j]);
    UserSendClient(WriteSockets.fd_array[j]);
    printf("**** End of this Out of Band packet\n");
   }
  ReadFlag = 0;
  FD_ZERO(&WriteSockets);
 }
```

Данная функция будет пытаться отправить внедиапазонные (Out-of-Band) данные на все гнезда в наборе *WriteSockets* fd_set. После отсылки данных fd_set очищается так, что больше уже не будет появляться в меню.

```
void CVisualDoc::OnInstallBH()
 {
  static int (CALLBACK * LastHook)();
    if (0==InstallBHFlag)
     {
       LastHook = WSASetBlockingHook(MyBlockingHook);

     if (NULL!=LastHook) InstallBHFlag = 1;
          else ErrorStop();

     }
```

```
  else
     { InstallBHFlag =0;
       WSASetBlockingHook(LastHook);
     }
}
```

Данная функция вызывается для установки или удаления *BlockingHook* (ловушка блокировки). Функция ловушки блокировки вызывается в том случае, когда блокируется функция гнезда. Целью ловушки блокировки является обработка сообщений в окне. Флаг *InstallBHFlag* показывает, установлена в данный момент ловушка блокировки или нет.

Если ловушка блокировки не установлена, то ее следует установить, вызвав функцию *WSASetBlockingHook*. Данная функция возвращает адрес предыдущей ловушки, если таковая была установлена ранее. Параметр предыдущей ловушки хранится в статической переменной *LastHook*.

Если *BlockingHook* уже присутствует, тогда вы устанавливаете ее снова в исходное значение, которое хранится в переменной *LastHook*. Переменная *InstallBlockingHookFlag* устанавливается таким образом, чтобы показывать, что ловушка блокировки не установлена.

```
/////////////////////////////////////////////////////////////////////////////
// CVisualDoc Flags Menu Commands

void CVisualDoc::OnLog()
 {
  LogFlag = (1==LogFlag)?0:1;
  dprintf("Log flag is now set to %d\n",LogFlag);
}
```

Данная функция вызывается для установки значения переменной *LogFlag*. Каждый раз после ее вызова текущее значение переменной *LogFlag* перебрасывается (изменяется на противоположное).

ОБНОВЛЕНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

Язык Visual C++ фирмы Microsoft обеспечивает механизм для обновления меню во время работы. Для обеспечения управления предполагаемым состоянием меню в этот пункт меню может быть введена специальная функция. Данная функция вызывается каждый раз, когда меню отображается на экране дисплея. Если пользователь щелкает мышью по раскрывающемуся меню, то до отображения меню осуществляется вызов функций. Это дает программе время принять решение о том, каким образом вывести на экран пункт меню.

Функции обновления UI имеют единственный аргумент (параметр) — указатель класса *CmdUI*. Данный класс содержит функцию элемента, которую вы можете вызвать для установки состояний меню. Основной функцией, используемой для внесения изменений в меню, является функция *SetCheck(int)*. Ниже приведены значения, которые может иметь функция *SetCheck*:

- 0 — обычное разрешенное меню;
- 1 — пометить меню галочкой;
- 2 — пометить меню точкой (.).

```
void CVisualDoc::OnUpdateLog(CCmdUI* pCmdUI)
 {
   pCmdUI->SetCheck(LogFlag);

 }
```

Данная функция обновляет *Log Menu* в раскрывающемся меню *Flags*. Она ставит галочку перед меню в том случае, если переменная *LogFlag* в данный момент содержит 1, либо она не ставит ничего в том случае, если *LogFlag* содержит 0. Смысл этого заключается в том, чтобы устанавливать галочку перед пунктом меню при разрешенном режиме регистрации.

```
void CVisualDoc::OnUpdateInstallBH(CCmdUI* pCmdUI)
 {
 pCmdUI->SetCheck(InstallBHFlag);

 }
```

Пункт меню *InstallBlockingHook* переключает установку *InstallBHFlag* таким же образом, как и *LogFlag* в предыдущей функции.

```
void CVisualDoc::OnUpdateWSAStartup(CCmdUI* pCmdUI)
 {
  if (WSAIsBlocking())
   {pCmdUI->Enable(0);
     return;
     }

     switch(WSAStartupCheck)
      { case(0):pCmdUI->SetCheck(0);
       return;
       case(1):pCmdUI->SetCheck(1);
            return;
       case(2):pCmdUI->SetRadio(1);
            return ;
   }
 }
```

Данная функция управляет режимом отображения пункта меню *WSAStartup* в раскрывающемся *Socket Functions*. Если в данный момент происходит блокировка внутри *Select* или в какой-либо другой функции гнезда, данный пункт меню будет деактивирован.

Вас может заинтересовать проблема, каким образом можно осуществить вызов функции, если вы заблокированы. Пользователь может установить *BlockingHook*. Если *BlockingHook* уже установлена, сообщения в окне будут продолжать обрабатываться, а если пользователь попытается раскрыть меню *Socket Functions*, то все функции Command UI в меню будут активированы, чтобы принять решение, каким образом отобразить их на экране. В этом случае пункт меню *WSAStartup* деактивирует себя путем вызова *enable()*.

Если вы не заблокированы, просмотрите глобальную переменную *WSAStartupCheck* и включите ее значение, чтобы принять решение, какой значок ставить — галочку или точку. Проще всего было бы создать следующую строку: pCmdUI- >SetCheck(WSAStartupCheck), однако оператор переключения игнорирует недействительные значения.

Алгоритм для Command UI, в принципе, тот же самый. Только переменная переключается после изменений:

```
void CVisualDoc::OnUpdateSocket(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
          return;
         }

         switch(SocketCheck)
          { case(0):pCmdUI->SetCheck(0);
                 return;
                 case(1):pCmdUI->SetCheck(1);
                      return;
                 case(2):pCmdUI->SetRadio(1);
                      return ;
          }
}

void CVisualDoc::OnUpdateBind(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
          return;
         }
```

```
        switch(BindCheck)
          { case(0):pCmdUI->SetCheck(0);
                return;
                case(1):pCmdUI->SetCheck(1);
                      return;
                case(2):pCmdUI->SetRadio(1);
                      return;
          }
}

void CVisualDoc::OnUpdateListen(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
         return;
        }

        switch(ListenCheck)
          { case(0):pCmdUI->SetCheck(0);
                return;
                case(1):pCmdUI->SetCheck(1);
                      return;
                case(2):pCmdUI->SetRadio(1);
                      return;
          }
}

void CVisualDoc::OnUpdateInsert(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
         return;
        }

        switch(InsertCheck)
          { case(0):pCmdUI->SetCheck(0);
                return;
                case(1):pCmdUI->SetCheck(1);
                      return;
                case(2):pCmdUI->SetRadio(1);
                      return;
          }
}

void CVisualDoc::OnUpdateInitialize(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
         return;
        }
```

```
 switch(InitializeCheck)
   { case(0):pCmdUI->SetCheck(0);
          return;
          case(1):pCmdUI->SetCheck(1);
               return;
          case(2):pCmdUI->SetCheck(0);
               return ;
   }
}
void CVisualDoc::OnUpdateSelect(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
          return;
         }
         switch(SelectCheck)
           { case(0):pCmdUI->SetCheck(0);
                 return;
                 case(1):pCmdUI->SetCheck(1);
                      return;
                 case(2):pCmdUI->SetRadio(1);
                      return;
           }
}
```

Следует заметить, что допустима привязка как Select, так и SelectRead к одной функции Command UI. Используйте ClassWizard и измените привязку SelectRead к OnUpdateSelectRead на привязку к OnUpdateSelect.

```
/////////////////////////////////////////////////////////////////////////////
// Actions UI Updates
void CVisualDoc::OnUpdateAccept(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
          return;
         }

   if ((2==InitializeCheck)&&(1==SelectCheck))
    {pCmdUI->SetCheck(0);
     return;
    }
   switch(ReadSockets.fd_count)
     { case(0):pCmdUI->SetCheck(0);
            return;
       default:if (FD_ISSET(ServerSocket,&ReadSockets)) pCmdUI->SetCheck(1);
                  else pCmdUI->SetCheck(0);
               return;
     }
}
```

Пункты в раскрывающемся меню Actions содержат несколько больше логики, чем требуется. Если клиент ожидает обработки, вам необходимо лишь появление галочки перед любой опцией. В случае accept проверьте следующие моменты:

- Вы не можете быть заблокированы в другой функции гнезда; если вы заблокированы, то отключитесь.
- Вы не можете просто инициализировать fd_sets; если да, то вы не установили контрольную метку (галочку). Инициализация FdSets характеризуется значением 2 для InitializeCheck и значением 1 для SelectCheck.
- Вам только необходимо поставить галочку, если в Read fd_set имеются гнезда (счет гнезд не равен 0), а гнездо сервера — это одно из гнезд.

Для других пунктов меню алгоритм тот же самый, однако, метка ставится пунктом меню, если в fd_set имеется хоть одно гнездо. Не играет роли, является ли данное гнездо гнездом *Main* (главным) или нет:

```
void CVisualDoc::OnUpdateOOBSend(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
          return;
         }

   if ((2==InitializeCheck)&&(1==SelectCheck))
    {pCmdUI->SetCheck(0);
          return;
         }
   if (0==WriteSockets.fd_count) pCmdUI->SetCheck(0);
   else pCmdUI->SetCheck(1);
}

void CVisualDoc::OnUpdateOOBRecv(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
          return;
         }

   if ((2==InitializeCheck)&&(1==SelectCheck))
    {pCmdUI->SetCheck(0);
          return;
         }

   if (0==ExceptSockets.fd_count) pCmdUI->SetCheck(0);
   else pCmdUI->SetCheck(1);
}
```

```
void CVisualDoc::OnUpdateSend(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
          return;
         }

   if ((2==InitializeCheck)&&(1==SelectCheck))
    {
     pCmdUI->SetCheck(0);
     return;
    }
   if (0==WriteSockets.fd_count) pCmdUI->SetCheck(0);
   else pCmdUI->SetCheck(1);
}

void CVisualDoc::OnUpdateRecv(CCmdUI* pCmdUI)
{
  if (WSAIsBlocking())
    {pCmdUI->Enable(0);
          return;
         }

  if ((2==InitializeCheck)&&(1==SelectCheck))
    {pCmdUI->SetCheck(0);
    return;
    }
  switch(ReadSockets.fd_count)
    { case(0):pCmdUI->SetCheck(0);
          return;
      case(1):if (FD_ISSET(ServerSocket,&ReadSockets)) pCmdUI->SetCheck(0);
                 else pCmdUI->SetCheck(1);
                    return;
          default:pCmdUI->SetCheck(1);
                return;

         }
    }
```

Если для пункта меню Recv счет гнезд составляет ровно 1 и это гнездо является гнездом сервера, то вам нет необходимости включать пункт меню *recv*, так как соответствующей операцией на гнезде сервера является *accept*.

```
//////////////////////////////////////////////////////////////////////////
// UI Commands for Variable fd_Sets Menu
```

Наиболее трудным меню, с точки зрения использования, является меню fd_sets, так как само меню должно видоизменяться. Для создания

нового пункта меню из существующего fd_set используется функция *MakeMenu.*

В данной функции создается пустое меню, а для каждого пункта в fd_set в конец меню добавляется (конкатенируется) новый пункт. Если в данный момент вы находитесь в функции блокировки, то пункт меню устанавливается в режим деактивирования, в противном случае он активирован. После выхода меню WINDOWS отключается от класса *CMenu* и возвращается. После завершения выполнения функции и ее выхода *CMenu m* уничтожается, однако, так как меню WINDOWS было отключено, то это меню не уничтожается.

```
HMENU CVisualDoc::MakeMenu(fd_set &fd,char c,UINT base)
  { CMenu m;
    HMENU rv;
    char buffer[80];
    unsigned int i,count=0;
    m.CreateMenu();
    for (i=0; i<fd.fd_count;i++)
          {

            sprintf(buffer,"%c(%d):Socket[%d]",c,count++,fd.fd_array[i]);
         m.AppendMenu(MF_STRING,base+i,buffer);
         if (WSAIsBlocking())
                        m.EnableMenuItem(base+i,MF_GRAYED);

         }

         rv = m.m_hMenu;
         m.Detach();
         return(rv);
  }

 BOOL CVisualDoc::OnCmdMsg(UINT nId, int nCode, void *pExtra,
 AFX_CMDHANDLERINFO *pHandlerInfo)
 {
 if ((20000<=nId) && (20500>nId))
   {
     if ((0==nCode)&&(NULL==pExtra)) ActivateSocket(nId);
    return(1);
   }
  if (ID_fdSetsHelp == nId)
   {
    if (-1 == nCode) return(CDocument::OnCmdMsg(nId,nCode,pExtra,pHandlerInfo));
   return(1);
   }
  else  return(CDocument::OnCmdMsg(nId,nCode,pExtra,pHandlerInfo));
 }
```

Это та самая функция, которая вызывается в том случае, когда командное сообщение любого типа посылается в класс. Возникла необходимость заменить эту функцию в данном классе для того, чтобы обработать fd_sets, которые будут иметь идентификаторы, присвоенные им во время работы. Если поступивший идентификатор находится между 20000 и 20500, то вызванная команда имеет место, так как гнездо в меню fd_set было выбрано. Данная функция опрашивает различные флаги, для того чтобы выяснить, является ли эта команда командой COMMAND или командой COMMAND UI, а затем вызывает соответствующие функции.

Если функция представляет собой COMMAND, [nCode = 0 and pExtra = NULL], тогда для активации соответствующего гнезда вызывается *ActivateSocket.*

Если функция представляет собой что-либо другое, то передайте команду на обычный *onCmd handler*, который можно найти в классе CDocument.

```
void CVisualDoc::ActivateSocket(UINT nId)
{
        char buffer[100];
        CWnd    *MainWindow;
        CMenu   *pMenu;
        char    c;
        int     cnt;
        SOCKET  iSocket;

        MainWindow = theApp.m_pMainWnd;
        pMenu = MainWindow->GetMenu();
        pMenu->GetMenuString(nId,buffer,99,MF_BYCOMMAND);

        sscanf(buffer,"%c(%d):Socket[%u]",&c,&cnt,&iSocket);
        switch(c)
         {
          case('A'): MainWindow->MessageBox("This menu choice is used"
                                  " to identify what sockets"
                                  " are available, and should"
                                  " not be selected","Warning",MB_OK);
                return;
          case('R'): if (ServerSocket == iSocket) OnAccept();
                else {if (0==UserReadClient(iSocket))
                      {dprintf("(%d) Socket Error - Socket being closed\n",iSocket);
                       FD_CLR(iSocket,&AllSockets);
                      }
                     FD_CLR(iSocket,&ReadSockets);
                     }
                return;
```

```
        case('W'): UserSendClient(iSocket);
                FD_CLR(iSocket,&WriteSockets);
                return;
        case('E'):ReadFlag = MSG_OOB;
                dprintf("**** (%d) Out of Band Data\n",iSocket);
                UserReadClient(iSocket);
                FD_CLR(iSocket,&ExceptSockets);
                dprintf("**** End of this Out of Band packet\n");
                ReadFlag = 0;
                return;
        }
}
```

Данная функция задействует *nID* для получения строки (string) Меню, а затем считывает информацию из этой строки в переменные. Первый символ строки идентифицирует, в каком fd_set находится гнездо. Если это 'A', то гнездо находится в наборе *All Sockets* fd Set. Сообщение распечатывается, указывая на то, что гнезда здесь не могут быть активированы.

Если этим символом является R, W или E, то значение гнезда считывается из строки и соответствующая функция обработки пользователя вызывается со значением гнезда.

```
void CVisualDoc::OnUpdatefdSets(CCmdUI* pCmdUI)
{
      CWnd *MainWindow;
      CMenu *pMenu;
      CMenu *pSubMenu;
       if (WSAIsBlocking())
         {pCmdUI->Enable(0);
          return;
         }

if ( 0 != pCmdUI-> m_nIndex) return;
MainWindow = theApp.m_pMainWnd;
pMenu = MainWindow->GetMenu();
pSubMenu = pMenu->GetSubMenu(1); // position of the fdset Menu
pSubMenu->DeleteMenu(4,MF_BYPOSITION);
pSubMenu->DeleteMenu(3,MF_BYPOSITION);
pSubMenu->DeleteMenu(2,MF_BYPOSITION);
pSubMenu->DeleteMenu(1,MF_BYPOSITION);
pSubMenu->AppendMenu(MF_POPUP,(UINT)MakeMenu(AllSockets,'A',ID_SOCKETITEM),"All Sockets");
pSubMenu->AppendMenu(MF_POPUP,(UINT)MakeMenu(ReadSockets,'R',ID_SOCKETITEM+100),"Read Sockets");
pSubMenu->AppendMenu(MF_POPUP,(UINT)MakeMenu(WriteSockets,'W',ID_SOCKETITEM+200),"Write Sockets");
pSubMenu->AppendMenu(MF_POPUP,(UINT)MakeMenu(ExceptSockets,'E',ID_SOCKETITEM+300),"Except Sockets");
return;
}
```

Данная функция вызывается для действительного создания меню *fd_sets* во время работы. Текущее меню *fd_sets* удаляется, а другое меню строится, в то время как пользователь пытается вывести на экран раскрывающееся меню.

Меню создано и содержит различные конкатенированные (добавленные ему в конец) субменю. Различные субменю создаются с помощью функции MakeMenu, передаваемой в соответствующий fd_set.

```
void CVisualDoc::OnfdSetsHelp()
{
    dprintf("This option is here to activate the following variable menus\n");
    dprintf("   All    Sockets\n");
    dprintf("   Read   Sockets\n");
    dprintf("   Write  Sockets\n");
    dprintf("   Except Sockets\n");

}
```

Это первое меню из меню *FdSets* и оно, как правило, деактивируется. Это ложное меню должно находиться там для того, чтобы было возможно различить COMMAND ID и COMMAND UI ID в функции *onCmd*.

РЕАЛИЗАЦИЯ ФУНКЦИЙ USERLIB (БИБЛИОТЕКИ ПОЛЬЗОВАТЕЛЯ)

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#undef WSAStartup
#undef socket
#undef bind
#undef listen
#undef accept
#undef connect
#undef recv
#undef send
#undef ioctlsocket
#undef select

int  CVisualDoc::UserLog_WSAStartup(WORD  VersionRequested, WSADATA*ImplementationDetailsPointer)
  {

   int err;

   if (1==LogFlag) dprintf("Calling WSAStartup(Version,Details) to locate a Windows DLL\n");

   err = WSAStartup(VersionRequested, ImplementationDetailsPointer);
```

```
  if (0!=err)
   {
     if (1==LogFlag){
        dprintf("      Couldn't locate a Windows Sockets DLL\n");
        dprintf("      WSAStartup Failed (Error Code = %d)\n",err);
   }
     return(err);
   }
  if (1==LogFlag)
  {
   dprintf(" WSAStartup has returned ImplementationDetails:\n");
   dprintf(" wVersion: (%ld)\n",ImplementationDetailsPointer- >wVersion);
   dprintf(" wHighVersion: (%ld)\n",ImplementationDetailsPointer->wHighVersion);
   dprintf("  szDescription:'%s'\n",ImplementationDetailsPointer- >szDescription);
   dprintf("  szSystemStatus:'%s'\n",ImplementationDetailsPointer->szSystemStatus);
   dprintf(" iMaxSockets: (%d)\n",ImplementationDetailsPointer- >iMaxSockets);
   dprintf(" iMaxUdpDg: (%d)\n",ImplementationDetailsPointer- >iMaxUdpDg);
   dprintf(" lpVendorInfo (%p)\n",ImplementationDetailsPointer- >lpVendorInfo);
  dprintf("\n\n");
  }
  return(0);
 }

SOCKET CVisualDoc::UserLog_socket(int AddressFormat,int Type,int Protocol)
 {
  SOCKET ServerSocket;
  if (1==LogFlag) dprintf("Calling socket() to Create a %s Socket\n",Me);

  ServerSocket = socket(AddressFormat,Type,Protocol);

  if (INVALID_SOCKET == ServerSocket) UserErrorMessage(" Socket creation Failed\n");
  else if (1==LogFlag) dprintf("  %s Socket (%d) has been created\n\n",Me,ServerSocket);

  return(ServerSocket);
 }

int CVisualDoc::UserLog_bind(SOCKET ServerSocket,LPSOCKADDR AddressPointer, int len)

 {
  int err;
  if (1==LogFlag) dprintf("Calling bind(Socket,AddressPointer,length) to Bind the socket\n");

  err = bind(ServerSocket,(LPSOCKADDR)AddressPointer,len);

  if (0 != err) UserErrorMessage("      Socket binding failed\n");
  else
    {if (1==LogFlag) dprintf("      Bind succeeded\n");
     UserShowAddress((LPSOCKADDR_IN)AddressPointer);
     if (1==LogFlag) dprintf("\n");
    }
```

```
  return(err);

 }

int CVisualDoc::UserLog_listen(SOCKET ServerSocket, int MaxConnections)
 {
   int err;
   if (1==LogFlag) dprintf("        Calling listen(Socket,MaxConnections)\n");

   err = listen(ServerSocket,MaxConnections);

   if (0 != err) UserErrorMessage(" Failed to Change to a listen Socket\n");
   else  if (1==LogFlag) dprintf("  Successfully changed to a listen socket\n\n");

   return(err);
 }

SOCKET CVisualDoc::UserLog_accept(SOCKET ListenSocket,LPSOCKADDR AddressPointer, int *lenPointer)
 {
   if (1==LogFlag) dprintf(" Calling accept(ListenSocket,AddressPointer,length)\n");
   if (1==LogFlag) dprintf(" This function will BLOCK until a client is available\n");

   ListenSocket = accept(ListenSocket,(LPSOCKADDR)AddressPointer,lenPointer);

   if (INVALID_SOCKET == ListenSocket) UserErrorMessage(" Socket accept failed\n");
   else if (1==LogFlag) dprintf(" accept succeeded: A client is available\n\n");

   return(ListenSocket);
 }

int CVisualDoc::UserLog_connect(SOCKET ServerSocket,LPSOCKADDR AddressPointer, int len)
 {
   int err;
   if (1==LogFlag) dprintf(" Calling connect(ServerSocket,AddressPointer,length)\n");
   UserShowAddress((LPSOCKADDR_IN)AddressPointer);

   err = connect(ServerSocket,(LPSOCKADDR)AddressPointer,len);

   if (0!= err) UserErrorMessage(     Socket connect failed\n");
   else if (1==LogFlag) dprintf("     Connect succeeded: A %s is available\n\n",NotMe);

   return(err);
 }

int CVisualDoc::UserLog_recv(SOCKET ClientSocket,char *buffer,int sBuffer,int flags)
 {
   int err;
   if (1==LogFlag) dprintf(" Calling recv(Socket,buffer,SizeOfBuffer,Flags)\n");

   err = recv(ClientSocket,buffer,sBuffer,flags);
```

```
    if (SOCKET_ERROR == err) UserErrorMessage(" Socket recv() failed\n");
    else
        if (0 == err) if (1==LogFlag) dprintf(" Zero Characters Read\n");
        else if (1==LogFlag) dprintf("%d characters read from the %s\n" "Data: %s\n",err,NotMe,buffer);

    return(err);
   }


int CVisualDoc::UserLog_send(SOCKET ClientSocket,char *buffer,int sBuffer,int flags)
  {
    int err;
    if (1==LogFlag) dprintf(" Calling send(Socket,buffer,SizeOfBuffer,Flags)\n");

    err = send(ClientSocket,buffer,sBuffer,flags);

    if (SOCKET_ERROR == err) UserErrorMessage(" Socket send() failed\n");
    else
        if (1==LogFlag) dprintf( "%d characters sent to the %s\n Data: %s\n",err,NotMe,buffer);

   return(err);
   }

void CVisualDoc::UserShowAddress(LPSOCKADDR_IN AddressPointer)
  {
     if (0==LogFlag) return;
     dprintf("        [AF_INET family is %d]\n",AF_INET);
     dprintf(" sin_family: %d\n",AddressPointer->sin_family);
     dprintf(" sin_port:   %d\n",AddressPointer->sin_port);
     dprintf(" sin_addr: %d.%d.%d.%d\n",
     AddressPointer->sin_addr.S_un.S_un_b.s_b1,
     AddressPointer->sin_addr.S_un.S_un_b.s_b2,
     AddressPointer->sin_addr.S_un.S_un_b.s_b3,
     AddressPointer->sin_addr.S_un.S_un_b.s_b4);
      }

int CVisualDoc::UserLog_ioctlsocket(SOCKET ServerSocket,long cmd, unsigned long *argptr)
   {
    int err;
    if (1==LogFlag) dprintf("Calling ioctlsocket(SOCKET,%lx,->[%lu]) to set non blocking mode",
       cmd,*argptr);
    err = ioctlsocket(ServerSocket,cmd,argptr);
    if (SOCKET_ERROR==err) UserErrorMessage("ioctlsocket failed\n");
    else if (1==LogFlag) dprintf("        ioctlsocket succeeded\n\n");
    return(err);
   }

 long CVisualDoc::UserLog_select(int ignore, fd_set *readfs, fd_set *writefs,
                        fd_set *errorfs, const struct timeval FAR * timeout)
   { long rv;
```

```
    CView   *pv;
    POSITION     pos;
    if (1==LogFlag)
    {
     dprintf("Calling Select to see if any activity is available\n");
     dprintf("  This call will BLOCK until at least one of the sockets is ready\n");
          dprintf("\n");
    }

         // since the select function may block, we must send a WM_PAINT
         // message to each view right now in order to cause the previous
         // dprintf statements to be visible while blocked...

        pos = GetFirstViewPosition();
        pv = GetNextView(pos);
         while(NULL!=pv)
          {
         pv->SendMessage(WM_PAINT,0,0);
         pv = GetNextView(pos);
         }
    rv = select(ignore,readfs,writefs,errorfs,timeout);
    if (SOCKET_ERROR == rv) UserErrorMessage("Select  Failed\n");
    else if (1==LogFlag)
     {
     dprintf("Select Returned (%ld)\n",rv);

     if (FD_ISSET(ServerSocket,&ReadSockets))
      {
       dprintf("A Client is ready to be attached on [Listening] Socket (%d)\n",ServerSocket);
      }

     UserExceptList(&ExceptSockets,&AllSockets);
     UserReadList(&ReadSockets,&AllSockets);
          }

    return(rv);
  }

void CVisualDoc::UserErrorMessage(char *message)
    {
     char buffer[100];
     char *Msg;
     int err;
     if (1==LogFlag) return;
     err = WSAGetLastError();
     dprintf("%s",message);
     switch(err)
      {
       case(WSAEINTR            ): Msg="WSAEINTR  ";
              break;
```

```
        case(WSAEBADF          ): Msg="WSAEBADF  ";
                break;
        case(WSAEACCES          ): Msg="WSAEACCES ";
                break;
        case(WSAEFAULT          ): Msg="WSAEFAULT ";
                break;
        case(WSAEINVAL          ): Msg="WSAEINVAL ";
                break;
       .
       .
       .

        case(WSANO_DATA          ): Msg="WSANO_DATA  ";
                break;
#endif
     default:sprintf(buffer,"       Unknown: Error (%d)",err);
          Msg=buffer;
          break;

     }

     dprintf("       Error:%s",Msg);
       }

void CVisualDoc::ErrorStop()
     {
      char buffer[100];
      char *Msg;
      int err;
      int stop=1;
      err = WSAGetLastError();
      switch(err)
       {
        case(WSAEINTR          ): Msg="WSAEINTR ";
                break;
        case(WSAEBADF          ): Msg="WSAEBADF ";
                break;
        case(WSAEACCES          ): Msg="WSAEACCES ";
                break;
        case(WSAEFAULT          ): Msg="WSAEFAULT ";
                break;
        case(WSAEINVAL          ): Msg="WSAEINVAL ";
                break;
        case(WSAEMFILE          ): Msg="WSAEMFILE ";
                break;
            .
            .
            .
```

```
        case(WSANO_RECOVERY ): Msg="WSANO_RECOVERY ";
                break;
        case(WSANO_DATA     ): Msg="WSANO_DATA ";
                break;
      default: sprintf(buffer,"Unknown: Error(%d)",err);
            Msg=buffer;
           break;

      }
     dprintf("Fatal Error: (%s)\n",Msg);
//     if (1==stop) exit(err);   // stop is currently disabled
    }

// ************************************** Connectionless Socket

SOCKET ConnectionlessSocket;
int CSocketFlag = 1;
int CBindFlag = 0;

void CVisualDoc::OnCreateConnectionlessSocket()
{

   if (2==CSocketFlag)
     dprintf("Warning:Already Created socket, should only create it once\n");

   ConnectionlessSocket = socket(PF_INET,SOCK_DGRAM,0);
   if (INVALID_SOCKET == ConnectionlessSocket) ErrorStop();
   CSocketFlag=2;
   CBindFlag = 1;

}

void CVisualDoc::OnBindConnectinlessSocket()
{  PortDialog input;

   if (2==CBindFlag)
      dprintf("Warning:Already called bind, should only call it once\n");

   input.m_port = 200;    // default port
   if (IDCANCEL == input.DoModal()) return;

   ServerSocketAddress.sin_family  = AF_INET;     // InterNet
   ServerSocketAddress.sin_addr.S_un.S_addr = INADDR_ANY;  // Any Address
   ServerSocketAddress.sin_port    = input.m_port;

rv = bind(ConnectionlessSocket,(LPSOCKADDR)&ServerSocketAddress,size of(ServerSocketAddress));
   if (0 != rv ) ErrorStop();

   CBindFlag=2;
}
```

```
int CPort=201;
int ci=0;
char INetAddress[80]="127.0.0.1";

void CVisualDoc::OnSendTo()
{
  char buffer[100],*data;
  SendToDial output;
  SOCKADDR_IN SocketAddress;

  sprintf(buffer,"Connectionless Message %d",ci);
  output.m_Data = buffer;
  output.m_Port = CPort;
  output.m_InterNet = INetAddress;
  if (IDCANCEL == output.DoModal()) {return;}

  CPort = output.m_Port;

  data = output.m_Data.GetBuffer(output.m_Data.GetLength());
  ci++;

   SocketAddress.sin_family  = AF_INET;     // InterNet
   SocketAddress.sin_addr.S_un.S_addr = inet_addr(output.m_InterNet);
   SocketAddress.sin_port    = output.m_Port;

   strcpy(INetAddress,inet_ntoa(SocketAddress.sin_addr));

  rv = sendto (ConnectionlessSocket,data,strlen(data),0, (LPSOCKADDR)&SocketAddress,sizeof(SocketAddress));

  If (SOCKET_ERROR == rv) ErrorStop();

}

void CVisualDoc::OnUpdateSendTo(CCmdUI* pCmdUI)
{
 // TODO: Add your command update UI handler code here

}

void CVisualDoc::OnRecvFrom()
{
  char buffer[100];
  SOCKADDR_IN SocketAddress;
  int AddressLength = sizeof(SocketAddress);

  rv = recvfrom(ConnectionlessSocket,buffer,sizeof(buffer),0, (LPSOCKADDR) &SocketAddress,&AddressLength);

  if (SOCKET_ERROR == rv)
    {ErrorStop();
       return;
      }
```

```
    dprintf("socket:%03d>[%03d]",ConnectionlessSocket,rv);
   output(rv,buffer);   // output is used because buffer may contain NUL characters

   UserShowAddress(&SocketAddress);

   CPort = SocketAddress.sin_port;
   strcpy(INetAddress,inet_ntoa(SocketAddress.sin_addr));

}

void CVisualDoc::OnUpdateRecvFrom(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here

}

void CVisualDoc::OnUpdateBindConnectinlessSocket(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here

}

void CVisualDoc::OnUpdateCreateConnectionlessSocket(CCmdUI* pCmdUI)
{

}
```

ОПИСАНИЕ КЛАССА CVISUALVIEW

Класс CVisualView

Класс *CVisualView* необходим для считывания информации из CVisualDoc и вывода ее на экран по требованию. Для описания данного класса с помощью AppWizard создается следующий файл VISUAVW.H:

```
// visuavw.h : interface of the CVisualView class
//
/////////////////////////////////////////////////////////////////////////////
class CVisualView : public CView
{protected: // create from serialization only
        CVisualView();
        DECLARE_DYNCREATE(CVisualView)
// Attributes
public:
        CVisualDoc* GetDocument();
// Operations
public:
```

```
// Implementation
public:
        virtual ~CVisualView();
        virtual void OnDraw(CDC* pDC);  // overridden to draw this view
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
        //{{AFX_MSG(CVisualView)
                // NOTE - the ClassWizard will add and remove member functions here.
                //   DO NOT EDIT what you see in these blocks of generated code !
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG        // debug version in visuavw.cpp
inline CVisualDoc* CVisualView::GetDocument()
  { return (CVisualDoc*) m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////
```

НОВЫЙ VISUAVW.H

```
// visuavw.h : interface of the CVisualView class
//
/////////////////////////////////////////////////////////////////////////////
//

class CVisualView : public CScrollView
```

Класс CVisualView является производным от CScrollView и наследует у него функциональные возможности прокрутки (scroll).

```
{
protected: // create from serialization only
        CVisualView();
        DECLARE_DYNCREATE(CVisualView)

// Attributes
public:
        int height;
        int width;
```

Эти атрибуты необходимы для поддержания возможностей прокрутки.

```
// Operations public:
        virtual void        OnInitialUpdate();
        virtual void        OnUpdate(CView *, LPARAM, CObject *);
```

Функции в классе *CView* не соответствуют данному классу и должны быть заменены.

```
        CVisualDoc* GetDocument();

// Implementation
public:
        virtual ~CVisualView();
        virtual void OnDraw(CDC* pDC);  // overridden to draw this view
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
        //{{AFX_MSG(CVisualView)
                // NOTE - the ClassWizard will add and remove member functions here.
                //    DO NOT EDIT what you see in these blocks of generated code !
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG          // debug version in visuavw.cpp
inline CVisualDoc* CVisualView::GetDocument()
  { return (CVisualDoc*) m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////
```

РЕАЛИЗАЦИЯ КЛАССА CVISUALVIEW

```
// visuavw.cpp : implementation of the CVisualView class
//

#include "stdafx.h"
#include "visual.h"

#include "visuadoc.h"
#include "visuavw.h"

#ifdef _DEBUG
```

```
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
//
// CVisualView

IMPLEMENT_DYNCREATE(CVisualView, CView)

BEGIN_MESSAGE_MAP(CVisualView, CView)
 //{{AFX_MSG_MAP(CVisualView)
      // NOTE - the ClassWizard will add and remove mapping macros here.
      // DO NOT EDIT what you see in these blocks of generated code !
 //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// CVisualView construction/destruction

CVisualView::CVisualView()
{
 // TODO: add construction code here
}

CVisualView::~CVisualView()
{
}

/////////////////////////////////////////////////////////////////////////////
//
// CVisualView drawing

void CVisualView::OnDraw(CDC* pDC)
{
        CVisualDoc* pDoc = GetDocument();

        // TODO: add draw code here
}
```

Данная функция будет заменена кодом программы в следующем сегменте.

```
/////////////////////////////////////////////////////////////////////////////
//
// CVisualView diagnostics

#ifdef _DEBUG
void CVisualView::AssertValid() const
```

```
{
        CView::AssertValid();
}

void CVisualView::Dump(CDumpContext& dc) const
{
        CView::Dump(dc);
}

CVisualDoc* CVisualView::GetDocument() // non-debug version is inline
{
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CVisualDoc)));
        return (CVisualDoc*) m_pDocument;
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CVisualView message handlers
```

НОВЫЙ VISUAVW.CPP

Данная функция заменяет предыдущую *OnDraw*.

```
void CVisualView::OnDraw(CDC* pDC)
{
 int LocNL;                          // Location of the NewLine
 int  isNewLineTerminated;           // Flag
 CVisualDoc*  pDoc = GetDocument();  // Pointer to current document
 POSITION  p;                        // POSITION for iteration
 CString  CurrentLine;               // String in Document
 int CurrentLineLength;              // length of the current  line
 TEXTMETRIC  tm;                     // Text information
 int  row=1;                         // What row are we drawing

 pDC->GetTextMetrics(&tm);
 height = tm.tmHeight;
 width  = tm.tmAveCharWidth·
```

Выполните цикл по всем строкам в текущем документе, распечатывая каждую строку в текущий контекст дисплея с помощью компонентной функции *TextOut* из класса CDC. Для каждой строки определите, является ли последний символ символом \n. В отличие от команды *printf, TextOut* будет действительно распечатывать символ 'пятно' (blob) для \n. Если *CurrentLine* оканчивается на \n, то вычтите единицу из ее длины, и *TextOut* не будет распечатывать \n.

```
for (p=pDoc->GetFirstStringPos();p !=NULL;)
  {CurrentLine = pDoc->GetNextString(p);
   LocNL = CurrentLine.GetLength()-1;
```

Возьмите строку *CurrentLine* и установите *LocNL* в 0, основываясь на том, где может располагаться \n.

```
  if (-1 == LocNL) isNewLineTerminated = 0;
  else    isNewLineTerminated = ('\n' == CurrentLine[LocNL]) ?-1 :0 ;
```

Если строка имеет нулевую длину, то она не оканчивается на \n. Если строка существует, проверьте визуально, является ли последний символ символом \n. Теперь *isNewLineTerminated* примет вид:

0 — если CurrentLine (текущая строка) не оканчивается на \n;
-1 — если CurrentLine оканчивается на \n.

Теперь вы можете присоединить *isNewLineTerminated* к длине *CurrentLine*, что автоматически вычтет единицу, если строка оканчивается на \n.

```
CurrentLineLength = CurrentLine.GetLength()+isNewLineTerminated;
```

Распечатайте текущую строку в контекст дисплея, а затем увеличьте ряд на 1. Заметьте, что ряд увеличится по высоте для правильного позиционирования текста в контексте дисплея. Команда *TextOut* стремится к позиционированию x и y в пикселах, однако вы имеете дело с символьными знаками, поэтому вам необходимо масштабирование по высоте символа.

```
   pDC->TextOut(0,row*height,CurrentLine,CurrentLineLength);
   row++;
  }
}

void CVisualView::OnInitialUpdate()
 {
  SetScrollSizes(MM_TEXT,GetDocument()->GetDocSize());
 }
```

Данная функция необходима для указания линейкам прокрутки размера документа, и она вызывается после инициализации документа. GetDocSize является компонентной функцией, заложенной в классе CVisualDoc.

```
void CVisualView::OnUpdate(CView *pSender, LPARAM lHint, CObject *pHint)
 {
  POINT Bottom;
  CSize OneLine(width,height);
  CSize temp = GetDocument()->GetDocSize();
```

```
  temp.cx *= width;
  temp.cy *= height;
   SetScrollSizes(MM_TEXT,temp,sizeDefault,OneLine);
  Bottom.x = 0;
  Bottom.y = temp.cy;
   ScrollToPosition(Bottom);
   CView::OnUpdate(pSender,lHint,pHint);
}
```

Данная функция необходима для указания линейкам прокрутки размера документа. Она вызывается каждый раз, когда документ обновляется. Единственный путь обновления — задействование функции элемента *dprintf*. Рассчитайте новое окончание документа и осуществите прокрутку до конца документа.

ПЕРВАЯ ПОПЫТКА РАБОТЫ В ПРОГРАММЕ-КОМПОНОВЩИКЕ (BUILD)

Если вы теперь осуществите компоновку, то вы получите следующий список ошибок:

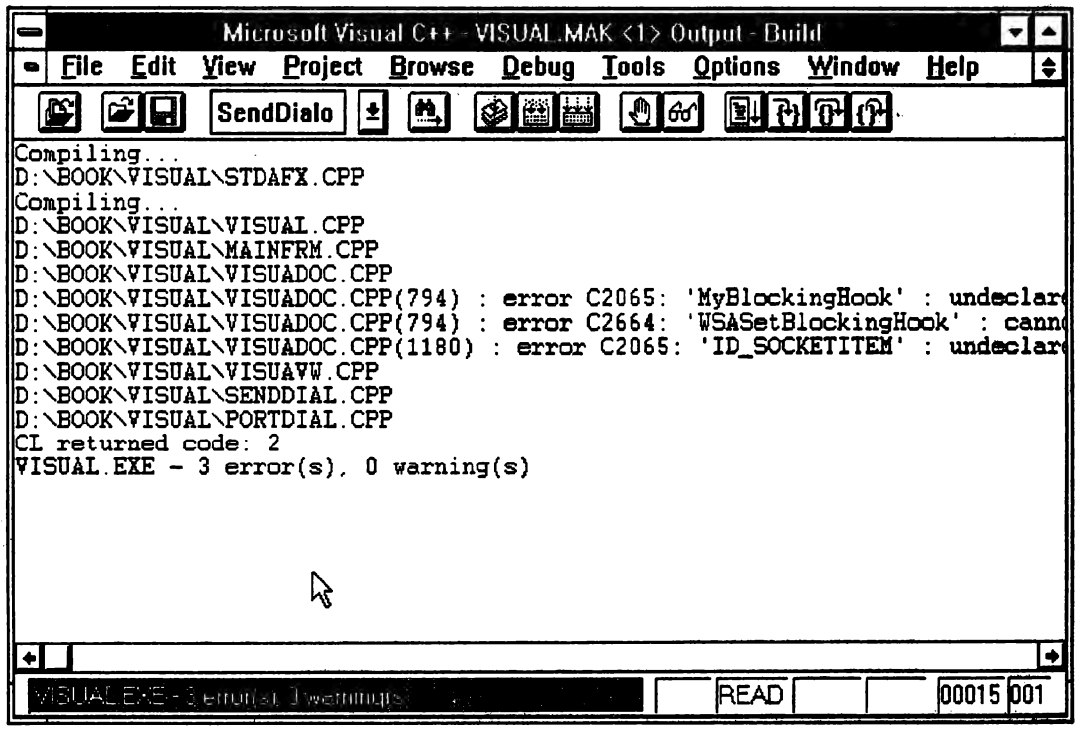


Рис. 9-44. Ошибки компоновщика.

Эти три ошибки отражают следующее:

- Вы не определили ловушку блокирования.
- Не определен символ ID_SOCKETITEM — он должен быть равным 20000.

Поместите определение для ID_SOCKETITEM в *resource.h*, список глобальных определений.

```
#define IDD_SendDialog        102
#define IDD_DIALOG1           103

#define ID_SOCKETITEM         20000

#define ID_WSAStartup         32768
#define ID_Socket             32769
#define ID_Bind               32770
```

СОЗДАНИЕ ЛОВУШКИ БЛОКИРОВАНИЯ

Добавьте следующую функцию в самый конец файла прикладной программы VISUAL.CPP. Этот файл содержит главный адрес входа в вашу программу и был создан с помощью AppWizard.

```
BOOL CALLBACK MyBlockingHook(void)
   {
    MSG msg;
    BOOL ret;
    /* get the next message if any */
     ret = (BOOL)PeekMessage(&msg,NULL,0,0,PM_REMOVE);
    /* if we got one, process it */
    if (ret) {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
    }
    /* TRUE if we got a message */
    return ret;
}
```

Вам необходимо добавить следующую строку кода программы для описания функции в контексте с *CVisualDoc Class*.

```
BOOL CALLBACK MyBlockingHook(void);
```

Эта строка должна быть добавлена в *VISUAL.h*.

ОСОБЕННОСТИ, ТРЕБУЮЩИЕ ВНИМАТЕЛЬНОГО ОТНОШЕНИЯ

Примите к сведению следующие возможные ошибки компилятора и компоновщика программ:

- Функция x уже содержит тело.

  Данная ошибка — результат наличия функции "заглушки" (stub function) x с каким-то телом, которое было создано некоторой программой-мастером (wizard), а вы обеспечили где-то еще второе тело. Отследите каждый случай наличия функции в вашем коде программы для выяснения того, можете ли вы найти эту заглушку. Это будет выглядеть следующим образом:

```
void CServerDoc::OnUpdateClearfdSets(CCmdUI* pCmdUI)
{
// TODO: Add your command update UI handler code here

}
```

- Функция x не является элементом класса y.

  Данная ошибка имеет место, потому что вы создали функцию с помощью ClassWizard, а затем сняли ссылку (reference) на эту функцию. Используя ClassWizard, внимательно посмотрите на идентификатор (ID), который обычно запускает функцию x. Возможно, вы заметите, что часть COMMAND или COMMAND UI класса отключена. Если это не так, то вы должны вручную удалить ссылку и осуществить повторное подключение.

  Для удаления подключения выберите кнопку delete. Появление блока сообщения означает, что вы должны удалить (delete) функцию. ClassWizard просто отменит подключение.

  Для повторного подключения осуществите функцию *add* снова. Будет создана новая функция заглушки, которую вам придется удалять из кода программы. Ваша предыдущая функция будет продолжать свое существование.

- Ошибка компоновщика: неразрешенная внешняя функция x.

  Эта ошибка — результат подключения к ClassWizard, однако, функция не определена в коде программы. Вы можете проверить это, попытавшись отредактировать код программы; ClassWizard сообщит вам, что функции не существует. Это происходит потому, что вы удалили функцию заглушки по ошибке. Для исправления данной ситуации используйте ClassWizard с целью удаления подключения и задействования его снова. Проигнорируйте предупреждение о необходимости удалить эту функцию вручную, так как ее в любом случае не существует. После повторного подключения функции для вас будет создана заглушка и вы сможете рекомпилировать и не рекомпоновать программу.

ОСУЩЕСТВЛЕНИЕ ФУНКЦИОНИРОВАНИЯ МЕНЮ FDSETS

Тот вид, в котором на данном этапе находится код программы, не позволяет еще работать меню *fd_sets*. Все это время все fd_sets находятся в деактивированном состоянии. Для того чтобы сделать их доступными, вы должны сделать следующие исправления ошибок в коде программы:

Расположите код программы:

```
ON_UPDATE_COMMAND_UI(ID_fdSETsHelp, OnUpdatefdSetsHelp)
```

в файле VISUADOC.CPP и измените его на:

```
ON_UPDATE_COMMAND_UI(ID_fdSETsHelp, OnUpdatefdSets)
```

Данный фрагмент кода программы был создан с помощью ClassWizard, когда вы подключили команду *ID_fdSetsHelp* к функции *OnUpdatefdSetsHelp*. Вы должны изменить отображение таким образом, чтобы оно ушло в *OnUpdatefdSets* для воссоздания меню, когда оно будет вызываться.

Если вы отслеживали поток кода программы, вы должны были заметить, что функция *OnUpdatefdSets* вызывалась пять раз, по одному разу для каждого пункта в меню fdSets. Элемент *m_nIndex* параметра *pCmdUI* будет принимать значения 0, 1, 2, 3 и 4. После того как вы проделали это, все меню устанавливаются в 0. Если *ID_fdSetsHelp* подключена к *OnUpdateSetsHelp*, меню считаются не созданными, т.к. вы не получаете вызова, когда индекс равняется 0. Данный вызов поступает на *OnUpdatefdSetsHelp*. Поэтому, для того чтобы меню заработало, вы должны вручную отредактировать ввод и поместить туда новое значение.

Глава 10

Программирование гнезд и интерфейсов транспортного уровня NetWare

ИНТЕРФЕЙС ГНЕЗД МЕЖСЕТЕВОГО/УПОРЯДОЧЕННОГО ОБМЕНА ПАКЕТАМИ NETWARE

Введение

Для написания в будущем прикладных программ клиент/сервер для NetWare необходимо заказать комплект программного обеспечения (SDK) для клиентов NetWare от фирмы NOVELL. Этот комплект содержит программное обеспечение и инструкции для написания программ пользователя в DOS, Windows и OS/2 в операционной среде NetWare версий 2.x, 3.x и 4.x.

NOVELL обеспечивает библиотеки для компиляторов Microsoft C (версии 6.0 и более поздние) и BORLAND (версии 3.0 и 3.1). Межсетевой и упорядоченный пакетный обмены (IPX и SPX соответственно) являются основой для набора протоколов NetWare. Межсетевой пакетный обмен (IPX) аналогичен межсетевому протоколу (IP) в межсетевом наборе протоколов; он представляет собой дейтаграммный уровень без логического соединения сетевого программного обеспечения, пакеты которого не требуют квитирования или упорядочения. Уровень SPX, стоящий непосредственно выше IPX, добавляет последнему виды сервиса, ориентированные на логическое соединение, обеспечивая надежную квитированную передачу данных. Взаимодействие SPX и IPX аналогично взаимодействию TPX и IP: SPX и TPX добавляют виды сервиса, ориентированные на логическое соединение, чисто дейтаграммному уровню. IPX обеспечивает совместимость с сетевыми системами фирмы Xerox Network Systems (XNS), с уровнем межсетевого дейтаграммного протокола (IDP), в то время как SPX обеспечивает совместимость с уровнем протокола упорядоченных пакетов (SPP) фирмы XNS.

Рассматривая данную проблему, желательно было бы обратиться к интерфейсу гнезд IPX и блоков управления событиями (ECB), которые используются программистами для обеспечения сопряжения с гнездами IPX. Необходимо также рассмотреть функции интерфейса транспортного уровня (TLI), которые скрывают гнезда интерфейса IPX от программиста TLI при использовании IPX для передачи данных. В примере программы используется TLI для программирования связи клиент-сервер фирмы Novell между двумя машинами с ОС DOS.

Адреса NetWare

Прежде чем программировать гнезда SPX, важно разобраться в адресах NetWare. Необходимо еще раз напомнить, что полный адрес в NetWare включает в себя 4-байтовый сетевой адрес, 6-байтовый адрес сетевого узла и 2-байтовый номер гнезда. Для доставки пакетов, когда адрес получателя находится на другом сетевом номере или физическом сегменте LAN, маршрутизаторы опираются на сетевую часть адреса, а не на адрес отправителя. Адрес сетевого узла однозначно идентифицирует узел PC как отправителя или получателя пакета на любом сетевом сегменте. При приеме пакетов драйвер сетевой информационной карты (NIC) распознает этот адрес на данном узле как свой и изымает пакет из сети.

Драйвер LAN передает пакет драйверу IPX, который, в свою очередь, передает его на обработку, связанную со специальным номером гнезда. В данном случае это является примером программирования "гнездо-гнездо" при упорядоченном пакетном обмене одного пользователя DOS с другим пользователем DOS через интерфейс транспортного уровня (Transport Layer Interface — TLI).

Блоки управления событиями

При передаче данных блоки управления событиями (Event Control Blocks ECB) представляют собой каналы связи между прикладной программой и межсетевым пакетным обменом (IPX). Прикладные программы передают уровень IPX кодовой информации в ECB, что необходимо для IPX при передаче или приеме пакетов.

Настройка ECB является основой программирования гнезд IPX. ECB могут быть гнездовыми или безгнездовыми: гнездовые ECB работают с событиями IPX/SPX, в то время как безгнездовые ECB функционируют на базе механизма синхронизации, что возможно благодаря синхронному планировщику событий. В данной книге рассматриваются только программы на основе гнезд (здесь и далее — "гнездовые" — прим. переводчика).

Поля ECB определены в структуре ECB файлом *nxtdh.h* для пользователей DOS и файлом *nxtw.h* для пользователей Windows (для получения более подробной информации см. Комплект программного обеспечения NetWare фирмы "Novell" (SDK NetWare)). Эти поля и их описание приведены ниже:

- **Адрес канала** (*link address*) — Данный адрес — *void far *linkAddress* — определяется как дальний указатель. В среде DOC дальний указатель — это 4-байтовый указатель, содержащий сегментный или регистровый базовый указатель и 16-битовое смещение или указатель регистра. Адрес поддерживается IPX при задействовании им ECB. Программист, при желании, может использовать этот адрес для поддержания свободного списка ECB.
- **Адрес программы обработки события (Event Service Routine (ESR) address)** — Данная переменная — *void (far *linkAddress)()* — определяется как дальний указатель для функции, вызываемый протоколом IPX, при использовании ECB для передачи или приема пакета. Если переменная не задействуется, то она устанавливается в ноль.

- **Задействованный флаг (in-use flag)** — IPX устанавливает данный флаг в ненулевое значение при задействовании ECB. По окончании задействования ECB для данного события протокол IPX сбрасывает эту переменную в 0. Существует семь возможных ненулевых значений, каждое из которых представляет возможное состояние обработки протокола IPX. Например, значение FE hex означает, что IPX находится в режиме прослушивания входящих пакетов на гнезде. В документации SDK фирмы Novell даются определения каждому значению *inUseFlag*.

- **Код завершения** (completion code) — IPX вводит значение в код завершения по окончании события, и ECB освобождается. Код завершения недоступен до тех пор, пока программные средства IPX не установят переменную *inUseFlag* в 0. В зависимости от типа события существуют шесть значений кода завершения. Ниже приведены четыре типа событий:

  — пересылка пакета,
  — ожидание пакета,
  — планирование асинхронного события,
  — вычеркивание события.

- Значение 0 кода завершения означает успешное завершение задачи.

- **Номер гнезда (socket number)** — Определяет номер гнезда, связанного с ECB и предыдущим вызовом *IPXOpenSocket()* (открытое гнездо IPX). Переменной является *socketNumber* (номер гнезда).

- **Рабочая область IPX (IPX workspace)** — Данная переменная, определяемая как *BYTE IPXWorkspace [4]*, является резервной. Она не может быть изменена, пока IPX задействует ECB.

- **Рабочая область драйвера (driver workspace)** — Данная переменная, определяемая как *BYTE driverWorkspace [12]*, также зарезервирована. Она не может быть изменена, пока IPX задействует ECB.

- **Адрес-операнд (immediate address)** — Представляет собой адрес последнего узла, который должен осуществить передачу пакета; т.е. адрес клиента, посылающего пакет, или маршрутизатор IPX. Этот элемент данных *BYTE immediateAddress [6]* — должен быть задействован для приема пакетов IPX.

- **Подсчет фрагментов (fragment count)** — Данная переменная, определяемая как *WORD fragmentCount*, представляет собой число буферных "фрагментов", составляющих пакет. Минимально это должна быть "1". Фрагменты представляют собой массив структур *ECBFragment*. Структура *ECBFragment* имеет следующий формат:

```
typedef struct ECBFragment
 { void far *address; WORD size;} ECBFragment;      //low-high (least- significant
                                                    //byte first, then
                                                    //most-significant byte)
```

Переменная указателя *address* является адресом фрагмента. Элемент *size* является размером фрагмента в виде ссылки указателя адреса.

- **Список дескриптора фрагментов ECB (ECB fragment descriptor list).** Данная переменная — *fragmentDescriptor [2]* — представляет две структуры *ECBFragment*, приведенные выше. Она определена как типовой *ECBFragment* в виде *ECBFragment fragmentDescriptor [2].*

Для получения более подробной информации о структуре данных ECB следует обратиться к SDK NetWare.

ИНТЕРФЕЙС ТРАНСПОРТНОГО УРОВНЯ (TLI)

Введение

Фирма Novell обеспечивает также интерфейс транспортного уровня (Transport Layer Interface — TLI) для IPX/SPX своего набора SDK. В качестве преамбулы к изучению представленного в данной книге программируемого интерфейса TLI мы проанализируем в общих чертах TLI, а затем рассмотрим особенности интерфейса TLI фирмы Novell для пользователей DOS.

TLI: Обзор

Интерфейс транспортного уровня (TLI), представленный в виде версии 3.0 ОС UNIX System V в 1986 году, также обеспечивает два режима сервисного обслуживания: на основе логического соединения и без логического соединения. Таким образом, по своим функциональным возможностям он аналогичен интерфейсу программирования гнезд Berkeley.

TLI обеспечивает интерфейс при определении уровня передачи данных при использовании протокола OSI. TLI в среде UNIX представляет собой набор задаваемых пользователем функций, что позволяет скрыть интерфейс *streams* от сетевой подсистемы. Разработку системы *Streams I/O* приписывают Деннису Ричи (Dennis Ritchie).

Терминология TLI отличается от терминологии, используемой для интерфейса гнезд BSD, хотя программирование обоих интерфейсов одинаково. В TLI две оконечные точки процесса взаимодействующих хостовых систем называются *transport endpoints* (оконечные точки передачи данных). В ОС UNIX они называются *sockets* (гнезда) или *half associations* (полуассоциации). *TLI Transport Provider* (средство обеспечения передачи данных) является транспортным протоколом — подобно набору протоколов TCP/IP — предоставляющим функции транспортного уровня. Хотя TLI обеспечивает *Interface* (интерфейс) при передаче данных, средство обеспечения передачи данных осуществляет реальную передачу данных по сети связи или поддержку переноса информации на обоих концах линии связи.

Ввод/вывод потоков данных

Для понимания TLI необходимо уяснить себе принцип потоков данных. Ричард Стивенс (W.Richard Stevens) описывает интерфейс потоков в работе *Unix Network Programming* (Prentice-Hall, Inc., 1990, pp. 374—378). Суммируя все сказанное выше, можно сказать, что потоки обеспечивают дуплексную (двустороннюю) связь между пользовательским процессом и драйвером устройства через *заголовок* (*head*) потока. В качестве драйвера устройства может выступать драйвер сетевой информационной карты (Network Interface Card — NIC), например Ethernet, Token Ring и т.д. Данный принцип изображен на рис. 10-1.

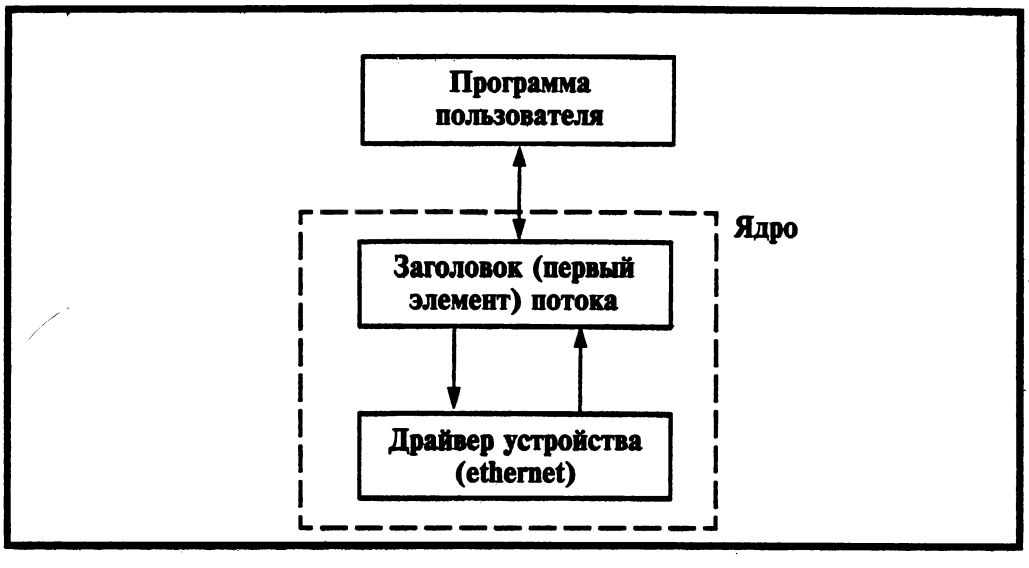


Рис. 10-1. Принцип потоков данных.

Потоки позволяют добавлять программные модули между первым элементом списка потоков, который является интерфейсом обращения к операционной системе по отношению к потокам, и драйвером устройства. Если этот модуль запрограммирован на прием информации от многочисленных источников, то он называется мультиплексором (*multiplexor*) потоков. Таким образом, с точки зрения функционирования сети, драйвер устройства потоков превращается в драйвер NIC, а программный модуль(и) между заголовком потока и драйвером NIC может реализовывать протокол связи, например такой, как TCP/IP. Данный принцип показан на рис. 10-2.

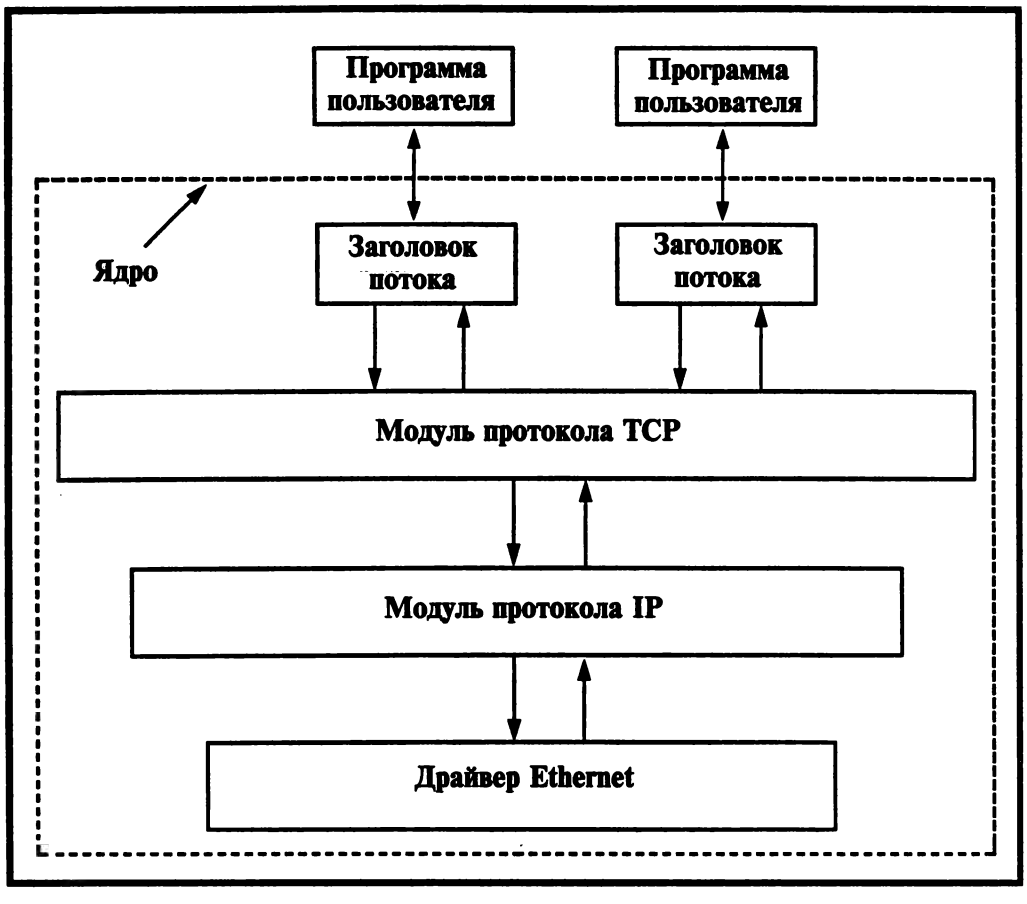


Рис. 10-2. Добавление программных модулей между заголовком потока и драйвером устройства.

Доступ к потокам осуществляется посредством следующих вызовов: *open, close, read, write, ioctl, outmsg, getmsg* и *poll.*

Повторное обращение к TLI

Библиотека функций TLI скрывает интерфейс потоков от пользователя, см. рис. 10-3.

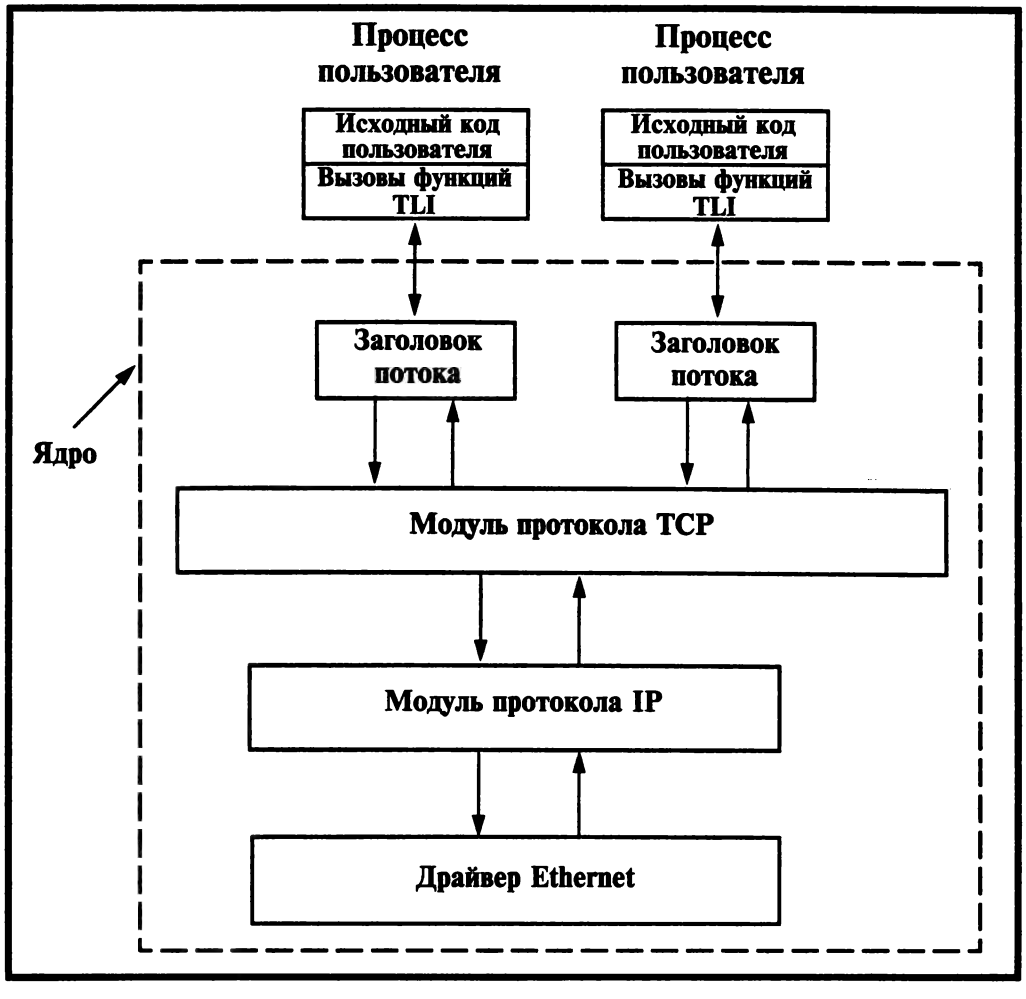


Рис. 10-3. Библиотека функций TLI скрывает интерфейс потоков от пользователя.

Реализация TLI фирмой Novell

Ввиду того что фирма Novell активно внедряется в среду UNIX, она приняла TLI в качестве стандарта для прикладных программ, которые требуют применения протокола транспортного уровня. В среде DOS интерфейс транспортного уровня используется в качестве библиотеки API, что позволяет обращаться к механизму протоколов IPX/SPX резидентной памяти ОС NetWare — обычно это IPX.COM клиента NetWare. Таким образом, TLI "прячет" программиста от интерфейса гнезда IPX/SPX.

TLI обеспечивает следующие преимущества по сравнению с прямым программированием гнезд IPX:

- **Многопротокольная поддержка (multiprotocol support)** — TLI прячет программу клиента от запросов выполнения протоколов сети. Прикладные программы, использующие TLI, требуют лишь незначительных изменений для того, чтобы соответствовать другим протоколам сети, таким, как TCP/IP.
- **Простота использования (ease of use)** — Неприятные тонкости, связанные с протоколами более низкого уровня, такими, как IPX, скрываются от пользователя. Поэтому прикладные программы TLI легче разрабатывать.
- **Стандартизация (standardization)** — Вместе с открытым интерфейсом транспортного уровня (XTI) TLI является принятым стандартом в операционной среде Unix.

Для получения более полной информации о преимуществах TLI см. документ «Инструкция программисту NetWare для языка "Си"» (NetWare Programmer's Guide for C), который входит в состав SDK клиентов NetWare.

Упорядочение вызовов клиент/сервер TLI

На рисунке 10-4 показан обычный порядок вызовов TLI между клиентом и сервером с использованием оконечных пунктов передачи данных, выполненных на основе логического соединения. Пример программы, представленный ниже в этой главе, характеризует TLI на основе логического соединения.

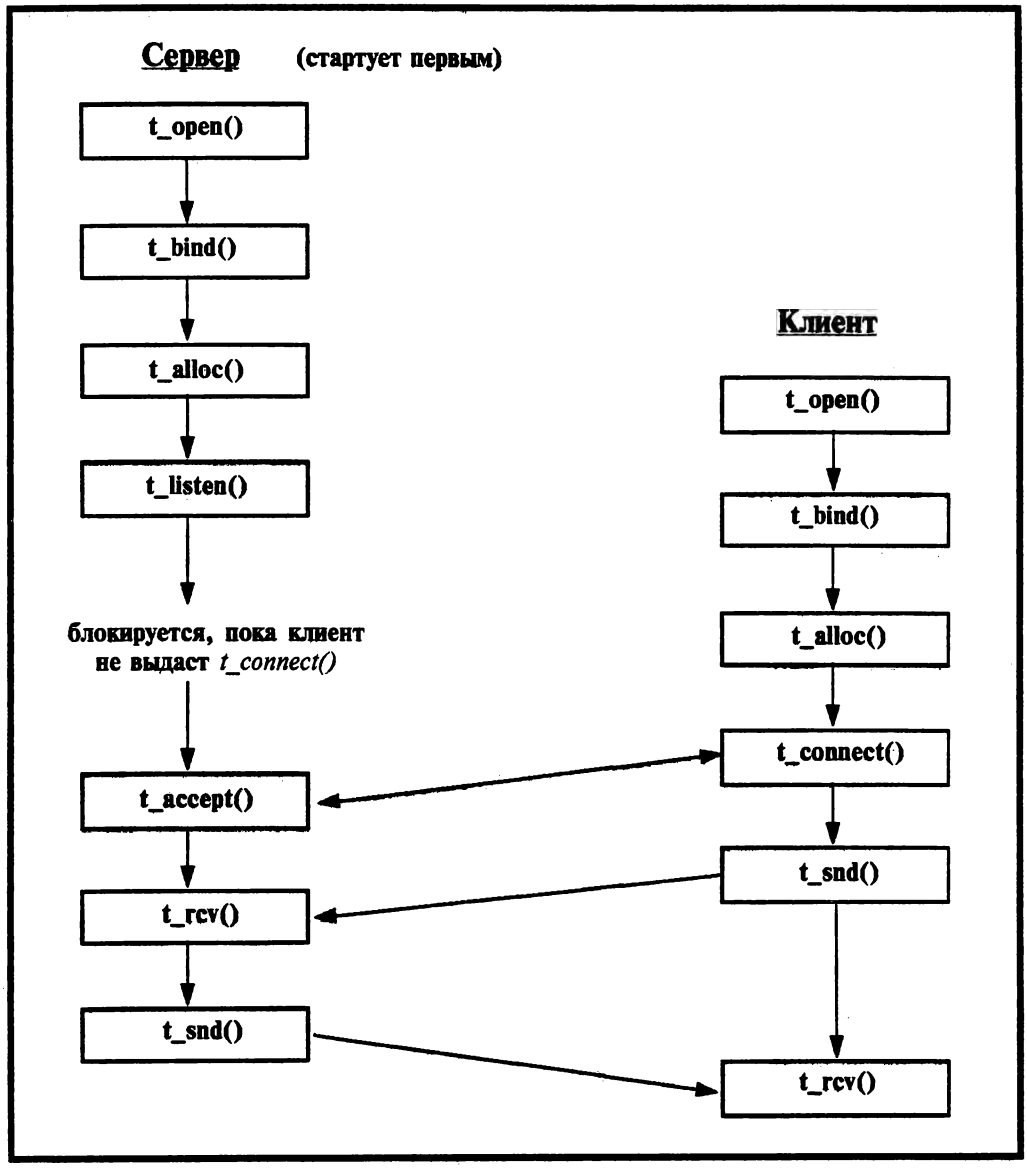


Рис. 10-4. Вызовы между клиентом и сервером с использованием оконечных пунктов передачи данных на основе логического соединения.

Структуры

TLI в реализации фирмы Novell определяет следующие типы структур параметров обработки для функций TLI:

- t_bind
- t_call
- t_optmgmt
- t_discon
- t_unitdata
- t_uderr
- t_info

Основной структурой для многопротокольной поддержки, при программировании TLI, является структура *netbuf*, имеющая следующий формат:

```
struct netbuf {
    unsigned int maxlen;
    unsigned int len;
    char         *buf;
};
```

Указатель *buf* указывает на символьный буфер, содержащий данные, в то время как *len* показывает количество символов, хранящихся в буфере в текущий момент. Указатель *maxlen* показывает максимальный размер буфера.

Большая часть типов структур TLI содержит указатели структур *netbuf*. Например, форматом структуры *t_bind* является:

```
struct t_bind {
    struct netbuf addr;
    unsigned  qlen;
};
```

Следует отметить, что переменная *addr* определяется как тип *struct netbuf*. Вложенность структуры *netbuf* внутри структуры *t_bind* позволяет выразить адрес в любом формате, который может потребовать локальное средство передачи данных. Подобная гибкость обеспечивается благодаря поддержке многопротокольной передачи данных. Данная структура имеет своей целью отделить TLI от свойственных только протоколу элементов данных, таких, как адреса. Например, адрес TCP/IP отличается от адреса IPX; однако оба протокола управления средствами передачи данных могут поддерживаться с помощью функций TLI.

Обзор функций TLI

Протокол управления средствами передачи данных (TLI) фирмы Novell включает в себя *t_alloc* и *t_free* для динамического распределения памяти и высвобождения места для переменных структуры TLI. Например, *t_alloc()* принимает следующий вид:

```
char *t_alloc(int fh, int struct_type, int fields);
```

Данная функция вызывается для распределения структуры. Входными параметрами являются: *fh*, или файловый описатель оконечного пункта, возвращенный из *t_open*; *struct_type*, который может быть одним из типов структуры, т.е. *struct t_bind*, *struct t_call* и т.д.; и *fields*, которые могут быть T_ADDR, T_OPT, T_UDATA и T_ALL. Для каждого из перечисленных выше полей *t_alloc* распределяет память для буфера, связанного с полем, и запускает *buf* и *maxlenl* со структурой *netbuf*, соответственно.

Большинство функций TLI обеспечивает поддержку режимов блокировки и разблокировки, очень напоминающих вызовы гнезд BSD. Оконечное устройство передачи данных управляет режимом блокировки. Сам режим выбирается в момент открытия программистом оконечного пункта. Функция *t_open()* может запросить режим разблокировки путем передачи флага O_NDE_LAY — для получения более подробной информации см. документацию SDK фирмы Novell. В противном случае оконечный пункт блокируется. Вызов *t_getinfo()* может обнаружить состояние блокировки оконечного пункта после его открытия.

t_open() принимает следующий вид:

```
#include <tiuser.h>
#include <nwipxspx.h>
#define NWWIN

int t_open
    (char        *path,
    int         oflag,
    struct t_info  *info);
```

где:

path указывает на имя пути доступа к файлу, который требуется открыть;

oflag определяет установки для флагов O_NDELAY и O_RDWR. Следует отметить, что оконечный пункт имеет режим считывания/записи;

info указывает на структуру, которая принимает информацию об используемом транспортном протоколе.

В качестве параметра ввода *t_open* принимает имя файла, идентифицирующего устройство обеспечения передачи данных, такого, как IXP или TCP/IP. Действительное имя для использования зависит от деталей применения задействованного устройства обеспечения передачи данных. С абонентским устройством обеспечения передачи данных TLI и IPX/SPX фирмы Novell вы можете, например, использовать */dev/nspx* для устройства передачи данных SPX ОС NetWare. Пример программирования дает дополнительную информацию.

Оператор *t_open* возвращает файл на обработку, для того чтобы использовать все функции последовательно для идентификации оконечного устройства передачи данных.

Структура *t_info*, указанная переменной *info*, имеет следующий формат:

```
struct t_info
{
    long addr;
    long options;
    long tsdu;
    long etsdu;
    long connect;
    long discon;
    long servtype;
} t_info
```

Данные параметры полей и их значения описаны в SDK фирмы Novell. Например, *addr* имеет значения, приведенные на рис. 10-5:

Поле	Значение	Смысл
addr	> 0	Максимальный размер адреса транспортного протокола
	— 1	Размер адреса без ограничений
	— 2	Устройство передачи данных не обеспечивает доступ пользователя к адресу протокола

Рис. 10-5. Значения адреса.

Рассмотрим библиотеку TLI фирмы Novell, построенную на базе библиотеки гнезд, описанной в программах, приведенных выше. После этого мы реализуем пример SPX на основе логического соединения клиент-сервер от одного клиента PC к другому клиенту PC, который реально действует, как сервер.

БИБЛИОТЕКА TLI ФИРМЫ NOVELL

Назначение

В данном примере показана упаковка протоколов TLI фирмы Novell в ту же библиотеку гнезд, которая описана в предыдущих примерах. Единственным отличием одной библиотеки от другой является внутренняя реализация различных функций элементов; внешнее использование библиотеки остается без изменений. Способом передачи данных в данном примере является упорядоченный пакетный обмен (SPX) фирмы Novell, протокол на основе логического соединения выше IPX, — а не TCP.

Библиотека CSOCKET

Имя библиотеки — CSOCKET, и она задействует строго определенные классы, которые использовались в предыдущей библиотеке CSOCKET:

- *CSocketBuf* — точно та же реализация;
- *CSocket* — головной файл и формат абсолютно идентичны, но здесь другая реализация элементных функций.

CSOCKET.CPP ФИРМЫ NOVELL (С ПОЯСНЕНИЯМИ)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "CSOCKET.H"

void startnet()
 {

 }

void stopnet()
 {
 }

void ProcessAsynchData(int x)
 {
 }
```

При совершении каких-либо событий включаются стандартные заголовочные файлы (header files) и формируется несколько функциональных заглушек (stubs), которые вызываются с помощью библиотеки класса.

```
int CSocket::count=0;
 int CSocket::flagdata=0;
 int CSocket::openMode= O_RDWR | O_NDELAY;
```

В дополнение к двум статическим элементам, описанным выше, вводится новый элемент, что позволяет обозначить используемый при открытии гнезда режим. Режим по умолчанию должен открывать гнездо Read/Write (считывание/запись) в неблокирующем режиме.

```
void output(char *buffer, char *end)
  {
  int count = end - buffer;
  while (count > 0)
    {
     if (31>*buffer) printf("\\%03d",*buffer);
     else   printf("%c",*buffer);

     buffer++;
     count—;
    }
  printf("\n");
  }

void output(char *buffer, int count)
  {
  while (count > 0)
    {
     if (31>*buffer) printf("\\%03d",*buffer);
     else   printf("%c",*buffer);

     buffer++;
     count—;

    }
  printf("\n");
  }
```

Два варианта функции вывода данных используются аналогичным образом, как и в библиотеке класса гнезд TCP/IP.

```
CSocketBuf::CSocketBuf()
 {
  unbuffered(0);
  allocate();
 }

CSocketBuf::doallocate()
 {
  int i = blen()+1000;
```

```
 char * p = new char [i];
 setb(p,p+i,1);
 setp(p+(i/2)+1,p+i);
 setg(p,p+(i/2)-1,p+(i/2)-1);
 return(1);
}

inline void CSocketBuf::setBuf(CSocket *in)
{
 SocketPointer = in;
}
```

Использование CSocketBuf идентично использованию библиотеки класса гнезд TCP/IP.

```
int CSocketBuf::overflow(int ch)
{
  if (SocketPointer->isLogging())  output(pbase(),pptr());
  if (SocketPointer->isConnected())
    {
     if (SocketPointer->isLogging())
         cout << "CSocketBuf:Sending" << endl;
     SocketPointer->send(pbase(),pptr()-pbase());
    }
  else
    {
     if (SocketPointer->isBound())
       {
/*       if (SocketPointer->isLogging())
           cout << "CSocketBuf:SendTo(" <<
           inet_ntoa(SocketPointer->LastAddress->sin_addr) << " port " <<
               SocketPointer->LastAddress->sin_port << endl;
           SocketPointer->sendto(pbase(),pptr()-pbase());
*/
       }
    else if (SocketPointer->isLogging())
      cout << "This socket is not bound and is not connected, can't send" << endl;
    }
  setp(pbase(),epptr());
  if (EOF!=ch) sputc(ch);
  return(1);
}
```

При переполнении библиотеки класса TCP/IP в некоторых случаях может быть вызвана функция *SendTo*. Данная часть кода программы содержится в виде комментария в библиотеке Novell, т.к. функциональность *SendTo* не реализована.

```
int CSocketBuf::underflow()
```

```
{
 char buffer[121];
 int l;
 if (gptr()!=egptr()) return(*gptr());
 if (SocketPointer->isConnected())
   {
    if (SocketPointer->isLogging())
      cout << "CSocketBuf:Recving" << endl;
    l = SocketPointer->recv(buffer,120);
   }
 else
   {
    if (SocketPointer->isBound())
       {
        if (SocketPointer->isLogging())
            cout << "CSocketBuf:recvfrom called" << endl;
        l =SocketPointer->recvfrom(buffer,120);
       }
    else
       {if (SocketPointer->isLogging())
          cout << "This socket is not bound and is not connected" << ", can't recvfrom" << endl;
        l=0;
       }
   }
 if (l<=0)
   {
    buffer[0]=32;
        buffer[1]=0;
    l=1;
   }
 buffer[l]=0;
 l = strlen(buffer);
 strncpy(egptr()-l,buffer,l);
 setg(eback(),egptr()-l,egptr());
 return(*gptr());
}

int CSocketBuf::sync()

{
 overflow(EOF);
 return(0);
}

CSocket::CSocket() : iostream(& sBuf)
{
 struct t_info TOpenInfo;
 flags        =flagdata;
 status       =0;
 Address      = new IPX_ADDR;
```

```
LastAddress   = new IPX_ADDR;
sBuf.setBuf(this);

if (0==count)
 {
  startnet();
 }
memset(&TOpenInfo,0,sizeof(TOpenInfo));
  s = t_open("/dev/nspx",openMode, &TOpenInfo);
```

Конструктор по умолчанию для CSocket используется аналогично конструктору TCP/IP, за исключением случая, когда задействуются функции и структуры TLI фирмы Novell. Структура *TOpenInfo* первоначально устанавливается во все нули. Функция *t_open* заполнит данную структуру информацией после создания гнезда TLI. Первый параметр к функции *t_open* представляет транспортный протокол, который будет использоваться. В библиотеке класса Novell используется протокол SPX. Статический элемент *CSocket:: openMode* используется для определения типа гнезда, которое необходимо создать. Если создаваемое гнездо является гнездом SERVER, *openMode* должна быть разблокирована. Если же это гнездо CLIENT, то *openMode* должна быть заблокирована. Разблокированный режим характеризуется использованием O_NDELAY.

```
if (INVALID_SOCKET == s) error("Creating Socket");
else
 {
  count++;
  if (isLogging())
   {
   cout << "Created socket (" << s << ") total sockets:" <<
   cout << endl ;
   cout << " TOpenInfo data" << endl;
   cout << "   addr:    " << TOpenInfo.addr << endl;
   cout << "   options: " << TOpenInfo.options << endl;
   cout << "   tsdu:    " << TOpenInfo.tsdu << endl;
   cout << "   etsdu:   " << TOpenInfo.etsdu << endl;
   cout << "   connect: " << TOpenInfo.connect << endl;
   cout << "   discon:  " << TOpenInfo.discon << endl;
   cout << "   servtype:" << TOpenInfo.servtype << endl;
   }
  }
}
```

Если создание гнезда TLI было успешным, то число гнезд увеличивается. Если разрешен режим регистрации, то распечатываются различные элементы структуры *TOpenInfo*.

```
CSocket::CSocket(int sock) : iostream(& sBuf)
```

```
{
 flags        =flagdata;
 status       =0;
 Address        = new IPX_ADDR;
 LastAddress    = new IPX_ADDR;
 sBuf.setBuf(this);
 s = sock;

 if (SOCKET_ERROR==sock)
  {
    if (isLogging())
     cout << "Error Socket Initialized" << endl;
    return;
  }

 if (0==count)
  {
   startnet();
  }
 count++;
 flags = flags | 0x05;
 if (isLogging())
   cout << "Initialized socket (";
   cout << s;
   cout << ") total sockets:";
   cout << count;
   cout << endl ;
}
```

Данный альтернативный конструктор используется в том случае, когда гнездо TLI уже было создано и оно должно быть вставлено в элемент *s* из *CSocket*, которое создается.

```
CSocket::~CSocket()
 {
  if (SOCKET_ERROR != s)
    {
    cout << "Flushing socket (" << s << ")" << endl;
    sync();
    }
  delete Address;
  delete LastAddress;
  if (SOCKET_ERROR == s)
    {
      if (isLogging())
         cout << "Error Socket closed" << endl;
      return;
    }
  count--;
```

```
 if (isLogging())
  cout << "Closing socket(" << s << ") " << endl;

   int rv = t_close(s);

 if (-1 == rv) error("closing socket");

 if (0==count)
   {
    stopnet();
   }
 if (isLogging())
  cout << "Deleted socket(" << s << ") total sockets:" << count <<
endl;

}
```

Деструктор для CSocket вызывает функцию *t_close*. Все остальное использование идентично классу TCI/IP.

```
CSocket & CSocket::operator = (CSocket &right)
 {
  if (SOCKET_ERROR != s)
    {
    cout << "Flushing socket (" << s << ")" << endl;
    sync();
    }
  if (SOCKET_ERROR == s)
    {
      if (isLogging())
          cout << "Error Socket closed" << endl;
    }
  else
    {
     if (isLogging())
     cout << "Closing socket(" << s << ") " << endl;

     int rv = t_close(s);
     if (-1 == rv) error("closing socket");
 }
  if (isLogging())
     cout << "Socket (" << s  << ") being replaced by (" << right.s <<
     ")" << endl;

  s = right.s;
  flags = right.flags;
  return(*this);
 }
```

```
BOOL CSocket::isLogging()
 {
  return(flags & 0x10);
 }

BOOL CSocket::isValid()
 {
  return(s!=SOCKET_ERROR);
 }
```

Все эти функции используются так же, как и их аналоги TCP/IP.

```
int CSocket::setLogging(int flag=1)
 {
  if (1==flag)
      flags = flags | 0x10;
  else flags = flags & 0xFEFF;

  return(0);
 }

BOOL CSocket::isBound()
 {
  return(flags & 0x01);
 }

BOOL CSocket::isListening()
 {
  return(flags & 0x02);
 }

BOOL CSocket::isConnected()
 {
  return(flags & 0x04);
 }
```

Большинство функций поддержки в классе *Csocket* не меняются при переходе к передаче данных в Novell.

```
int CSocket::bind(unsigned short port, IPXADDR_CHAR addr)
 {
   char uf[1000];
   if (SOCKET_ERROR == s)
     { if (isLogging())
         cout << "Can't Bind an error socket" << endl;
       return(-1);

     }
   if (isBound())
```

```
    { if (isLogging())
        cout << "Socket " << s << " Already Bound" << endl;
        return(-1);
    }
   if (NULL!=addr)
{
    if ( (strlen(addr) !=21) || (':' != addr[8]) )
     { cout << " Invalid IPX Net/Address Specified" << endl;
       cout << "   XXXXXXXX:XXXXXXXXXXXX, with leading 0s if required" << endl;

       return(-1);
     }
    int sc = sscanf(addr,"%2x%2x%2x%2x:%2x%2x%2x%2x%2x%2x",
        &Address->ipxa_net[0],
        &Address->ipxa_net[1],
        &Address->ipxa_net[2],
        &Address->ipxa_net[3],
        &Address->ipxa_node[0],
        &Address->ipxa_node[1],
        &Address->ipxa_node[2],
        &Address->ipxa_node[3],
        &Address->ipxa_node[4],
        &Address->ipxa_node[5]);

 if (10 != sc)
 {
cout << " Invalid format for address.  Only recognized " << sc << endl;
cout << " of the 10 bytes being read.  Format is BBBB:BBBBBB, where" << endl;
cout << " each B represents two HEX digits" << endl;
return (-1);
 }
}
```

Новый код программы используется здесь для сканирования адреса SPX в двоичный эквивалент.

```
   Address->ipxa_socket[0]        = 0;
   Address->ipxa_socket[1]        = (int) port;

    memset(LastAddress,0,sizeof(IPX_ADDR));

   struct t_bind t_input;
   struct t_bind t_output;
   t_input.addr.maxlen = sizeof(IPX_ADDR);
   t_input.addr.len    = sizeof(IPX_ADDR);
   t_input.addr.buf    = (char *) Address;
   t_input.addr.maxlen = sizeof(IPX_ADDR);
   t_input.addr.len    = sizeof(IPX_ADDR);
   t_input.addr.buf    = (char *) Address;
   t_input.qlen        = 5;    // allow up to 5 clients
```

```
t_output.addr.maxlen = sizeof(IPX_ADDR);
t_output.addr.len    = sizeof(IPX_ADDR);
t_output.addr.buf    = (char *) LastAddress;
int rv;

if (NULL==addr)
    rv = t_bind(s,NULL,&t_output);
else rv = t_bind(s,&t_input,&t_output);
```

Функция *t_bind* позволяет привязать гнездо к локальному адресу. Если вам безразлично, к какому адресу вы привяжетесь, то вы можете перейти в NULL (НОЛЬ) в качестве входного аргумента. Функция принимает три аргумента: предыдущее открытое гнездо *t_open*'ed *socket*, входную структуру *t_bind* и выходную структуру *t_bind*.

Структура *t_bind* используется для определения адреса, к которому гнездо должно быть привязано и к которому оно действительно привязано. Структура *t_bind* содержит два элемента, в которых заинтересован пользователь в ходе привязки. Элемент *addr* используется для указания на зависящую от протокола структуру адреса, а элемент *qlen* содержит такое количество клиентов, которое можно поставить в очередь для обслуживания с помощью сервера. Элемент *addr* содержит три дополнительных элемента: *maxlen*, *len* и *buf*. Элементы *maxlen* и *len* используются для указания длины буфера, которая помещается в элементе *buf*. Ввиду того что гнездо было открыто с помощью протокола SPX, элемент *addr.buf* должен указывать на структуру IPX_ADDR, которая является основой адреса SPX.

```
if (-1 == rv) return(error("Binding socket"));
else flags = flags | 0x03;  // set listen flag as well

if (isLogging())
 {
 Bcout << "Binding(" << s  << ") to " << addr << " port " << port << endl;

 cout << "Socket is bound to:" << endl;

sprintf(buf,"  Net:%X %X %X %X\n  Node:%X %X %X %X %X %X\n  Socket: %X %X",
    LastAddress->ipxa_net[0],   LastAddress->ipxa_net[1],
    LastAddress->ipxa_net[2],   LastAddress->ipxa_net[3],
    LastAddress->ipxa_node[0],  LastAddress->ipxa_node[1],
    LastAddress->ipxa_node[2],  LastAddress->ipxa_node[3],
    LastAddress->ipxa_node[4],  LastAddress->ipxa_node[5],
    LastAddress->ipxa_socket[0],LastAddress->ipxa_socket[1]);
   cout << buf << endl;
  }
 return(0);
}
```

После вызова функции *t_bind* значение возврата проверяется на предмет возникновения ошибки. Если гнездо было привязано успешно, то информация в буфере *t_output* распечатывается, при условии если разрешен режим регистрации. Буфер *t_output* содержит указатель на *LastAddr* структуры IPX_ADDR, которая содержит адрес SPX, возвращенный с помощью *t_bind*.

```
int CSocket::unbind()
 {
  Bif (isBound())
    { int rv = t_unbind(s);
      if (-1 == rv) return(error("Unbinding socket"));

      if (isLogging())
       cout << "Unbinding(" << s << ") was successful" << endl;
    }
  else cout << "Can't unbind unless the socket is bound" << endl;
  return(0);
 }
```

Функция *unbind* не реализуется в библиотеке класса TCP/IP, т.к. эти два протокола не обеспечивают функциональных возможностей по отмене привязки гнезд. Применение TLI фирмы Novell обеспечивает реальную возможность отменять привязку любого гнезда; таким образом, реализуется компонентная функция *unbind*.

```
int CSocket::listen(int queue)
 {
   if (SOCKET_ERROR == s)
    { if (isLogging())
      cout << "Can't Listen on an error socket" << endl;
      return(-1);

    }
   if (isListening())
    { if (isLogging())
        cout << "Socket " << s << " Already Listening" << endl;
        return(-1);
    }

//   int rv = ::listen(s,queue);
//   t_listen is implemented differently on TLI; it is enabled at t_bind

//   if (-1 == rv) return(error("Listening..."));
//   else flags = flags | 0x02;

   if (isLogging())
    cout << "socket " << s << " is now listening" << endl;
   return(0);
 }
```

Функция *Listen* обеспечивается для режима эмуляции с библиотекой TCP/IP. Функциональная возможность режима прослушивания для входящего запроса в *t_bind* разрешается автоматически.

```
int CSocket::StopListening()
 {
  if (isLogging())
   cout << "Stop Listening Not supported in this implementation" << endl;
   return(-1);
 }

CSocket & CSocket::accept()
 {
  CSocket *cs;
  SPX_OPTS spx_options;
  IPX_ADDR spxAddress;
  struct t_call InRequest;
  char udata[80];

  int size=sizeof(IPX_ADDR);

  if (SOCKET_ERROR == s)
   { if (isLogging())
       cout << "Can't accept on an error socket" << endl;
     return *this;
   }
  if (isLogging())
    cout << "Calling accept to get a client, this may block" << endl;
```

Компонентная функция *accept* объединяет несколько функций TLI вместе, для того чтобы обеспечить функциональные возможности, подобные TCP/IP. Компонентная функция *accept* используется для обеспечения серверу возможности подождать, пока клиент не подключится. Эта функция осуществляет возврат либо гнезда, которое подключено к клиенту, либо ошибочного гнезда, в случае ошибки. Данная компонентная функция обеспечивает следующее:

Она находится в режиме прослушивания гнезда до тех пор, пока к нему не попытается подключиться клиент. Гнездо сервера должно быть открыто и находиться в неблокирующем состоянии. В противном случае **t_listen** зависнет. До тех пор пока функция **t_listen** не вернет TNODATA, программа будет зациклена до момента, когда данные становятся доступными. При поступлении на гнездо внедиапазонной информации ошибка возвращается с помощью TLOOK, и вы должны обработать асинхронное сообщение. В случае попытки клиента подключиться его адрес помещается в *InRequest* структуры *t_call*.

После того как сервер подождал и обнаружил клиента, с помощью функции **t_open** создается новое гнездо. Это гнездо привязывается к локальному неуточненному адресу с использованием функции **t_bind**, а локальный адрес не возвращается обратно, т.к. значение NULL было пройдено как третий параметр.

После создания нового гнезда и его привязки к локальному адресу, вызывается функция **t_accept** для того, чтобы подсоединить только что созданное гнездо к клиенту, который пытается подключиться. После возвращения функции **t_accept** новое гнездо может быть использовано для связи с клиентом, а исходное гнездо сервера становится доступным для подключения дополнительного клиента.

```
memset(udata,0,80);
memset(&spx_options,0,sizeof(SPX_OPTS));

InRequest.addr.maxlen=sizeof(IPX_ADDR);
InRequest.addr.len   =sizeof(IPX_ADDR);
InRequest.addr.buf = (char *) &spxAddress;
InRequest.opt.maxlen =sizeof(IPX_OPTS);
InRequest.opt.len    =sizeof(IPX_OPTS);
InRequest.opt.buf  = (char *) &spx_options;
InRequest.udata.maxlen = 0;
InRequest.udata.len    =0;
InRequest.udata.buf = (char *) 0;
```

Структура *t_call* должна быть активирована с помощью определенных указателей до вызова **t_listen**. Исходные значения *InRequest* структуры *t_call* указывают на пустые адреса IPX_ADDR и структуры опций IPX_OPTS. Любые дополнительные данные пользователя отвергаются, т.к. элементы udata установлены в 0.

```
int rv = -1;
while(-1 == rv)
 { rv = t_listen(s,&InRequest);
  if (-1 == rv)
    {switch(t_errno)
     {
      case(TLOOK): ProcessAsynchData(s);
      case(TNODATA): break;
      default:
        error("Listening at Socket");
        cs = new CSocket(SOCKET_ERROR);
        return(*cs);
     }
    }
 }
```

Это — часть кода программы, где вы ожидаете подключения клиента. Каждый раз при вызове функции **t_listen** она возвращается немедленно, т.к. гнездо сервера было создано с использованием O_NDELAY. Если значение **t_listen** равно -1, то вам необходимо выяснить, какой вид ошибки имел место.

TNODATA — эта ошибка показывает, что у **t_listen** не было готового клиента.

TLOOK — эта ошибка показывает, что вам необходимо просмотреть *t_error* для получения более подробной информации об ошибке. Это означает, что асинхронные данные могут быть доступны и вы должны их обработать. Эта реализация игнорирует все асинхронные данные.

Если значение возврата НЕ равно -1, то это значит, что **t_listen** обнаружила клиента, и информация от него помещена в структуру *InRequest t_call.* Фрагмент кода программы на следующей странице осуществляет распечатку информации в структуре *t_call,* при условии если разрешен режим регистрации.

```
  if (isLogging())
   {
   char buf[1000];
   cout << "A client is trying to connect" << endl;
   cout << " sequence:  " << InRequest.sequence << endl;
   cout << " addr.len:  " << InRequest.addr.len << endl;
   cout << " address: " << endl;

sprintf(buf,"   Net:%X %X %X %X\n  Node:%X %X %X %X %X %X\n  Socket: %X %X",
     spxAddress.ipxa_net[0],   spxAddress.ipxa_net[1],
     spxAddress.ipxa_net[2],   spxAddress.ipxa_net[3],
     spxAddress.ipxa_node[0],  spxAddress.ipxa_node[1],
     spxAddress.ipxa_node[2],  spxAddress.ipxa_node[3],
     spxAddress.ipxa_node[4],  spxAddress.ipxa_node[5],
     spxAddress.ipxa_socket[0],spxAddress.ipxa_socket[1]);
    cout << buf << endl;

    cout << "  opt.len:  " << InRequest.opt.len << endl;
    output((char *) &spx_options,InRequest.opt.len);
    cout << "  udata.len:" << InRequest.udata.len << endl;
    output(udata,InRequest.udata.len);

    cout << endl<< "Attempting to accept the client" << endl;
    }

  struct t_info OpenInfo;
  memset(&OpenInfo,0,sizeof(OpenInfo));
```

```
int sock = t_open("/dev/nspx",O_RDWR,&OpenInfo);
if (-1 == sock)
  {
   error("Creating the server endpoint");
   cs = new CSocket(SOCKET_ERROR);
   return(*cs);
  }

rv = t_bind(sock, (struct t_bind *) 0,(struct t_bind *) 0);
if (-1 == rv)
  {
   error("Binding the server endpoint");
   t_close(sock);
   cs = new CSocket(SOCKET_ERROR);
   return(*cs);
  }
```

Данный сегмент кода программы создает новое гнездо и привязывает его к локальному адресу на сервере. После выполнения данного сегмента кода программы доступными становятся два гнезда. Исходное гнездо сервера все еще доступно для клиентов, пытающихся подключиться, и имеется новое гнездо, которое было только что образовано. Оно будет использовано для обеспечения связи с клиентом, который пытается получить доступ. Данный сегмент кода программы создает новое гнездо и привязывает адрес к нему. Однако новое гнездо еще не соединено с клиентом.

```
rv = t_accept(s,sock,&InRequest);
  if (-1 == rv)
    {
     if (t_errno == TLOOK) ProcessAsynchData(s);
     else
       {
        error("Accepting a client");
        cs = new CSocket(SOCKET_ERROR);
        return(*cs);
       }
    }

if (isLogging())
  cout << "A new Client is accepted" << endl;

cs = new CSocket(sock);
return(*cs);
}
```

Функция **t_accept** вызывается сервером для подключения местного оконечного пункта — гнезда, которое было создано в предыдущем шаге — к удаленному оконечному пункту, указанному структурой *t_call,*

пропущенной в качестве третьего параметра. Первый параметр — это гнездо, которое клиент использовал для подключения к серверу. В случае возникновения ошибки может последовать дополнительная обработка. Если *t_accept* проходит, то клиент соединяется с новым оконечным пунктом и он теперь может использоваться для связи. Новый класс CSocket C++ создается путем использования специального конструктора, который формирует CSocket в момент, когда гнездо сети уже существует. Следует отметить, что обычный конструктор создает гнездо сети во время конструирования, однако, ввиду того, что гнездо сети уже существует, обычный конструктор не может быть использован. Значение возврата, принадлежащее компонентной функции *accept*, это CSocket, и именно это значение подлежит возврату.

```
int CSocket::connect(unsigned short port,IPXADDR_CHAR addr)
 {
  SPX_OPTS  spx_options;

  char buf[1000];
  if (SOCKET_ERROR == s)
   { if (isLogging())
        cout << "Can't Connect on an error socket" << endl;
     return(-1);

   }
  if (isConnected())
   { if (isLogging())
       cout << "Socket " << s << " Already Connected" << endl;
       return(-1);
   }
```

Это функция, которая используется клиентом для связи с сервером. Двумя параметрами, которые должны передаваться в функцию, являются: адрес сети сервера и специальный порт на узле сервера, где программа сервера будет находиться в режиме прослушивания. Адрес сети сервера аналогичен физическому адресу карты сети, т.е., адресу локальной сети Ethernet карты Ethernet. Как часть системы, запускающей программу сервера, адрес сети, к которой сервер был привязан, может быть распечатан при условии, если разрешен режим регистрации. Это должен быть тот же адрес, что и переданный в функцию соединения. Формат адреса содержит всего 12 байтов: первые четыре используются для идентификации номера сети фирмы Novell, а остальные восемь — это физический адрес сети. Форматом является:

NNNNNNNN:PPPPPPPPPPPPPPP,

где каждая N представляет шестнадцатиричную цифру в сети Novell, а P представляет шестнадцатиричную цифру в физическом адресе.

Типичным адресом может быть выражение:

00000000:0080C7B20602,

считанное со следующей распечатки сервера:

Server is bound to:	Сервер привязан к:
Net:0 0 0 0	Сеть:0 0 0 0
Node:0 80 C7 D2 6 2	Узел:0 80 C7 D2 6 2
Socket: 0 64	Гнездо: 0 64

Следует отметить, что все байты должны иметь начальные нули, добавленные, при необходимости, в адрес. Значение гнезда 0 64 представляет порт 100 (десятичный).

```
if ( (strlen(addr) !=21) || (':' != addr[8]) )
  { cout << " Invalid IPX Net/Address Specified" << endl;
    cout << "   XXXXXXXX:XXXXXXXXXXXX, with leading 0s if required" << endl;
    return(-1);
  }

 int sc = sscanf(addr,"%2x%2x%2x%2x:%2x%2x%2x%2x%2x%2x",
       &Address->ipxa_net[0],
       &Address->ipxa_net[1],
       &Address->ipxa_net[2],
       &Address->ipxa_net[3],
       &Address->ipxa_node[0],
       &Address->ipxa_node[1],
       &Address->ipxa_node[2],
       &Address->ipxa_node[3],
       &Address->ipxa_node[4],
       &Address->ipxa_node[5]);

 if (10 != sc)
  {
 cout << " Invalid format for address.  Only recognized " << sc << endl;
 cout << " of the 10 bytes being read.  Format is BBBB:BBBBBB, where" <<
 endl;
 cout << " each B represents two HEX digits" << endl;
 return (-1);
  }
```

Эта часть кода программы осуществляет сканирование строки IPXADDR_CHAR в формате:

NNNNNNNN:PPPPPPPPPPPP

в структуру IPX_ADDR, указанную адресом переменной. Если строка адреса имеет несоответствующую длину, или ":" стоит на неправильном

месте, или количество считанных элементов данных не равно 10, то это вызывает ошибку, означающую, что строка IPXADDR_CHAR недопустима. После успешного завершения выполнения данного сегмента кодировки элемент адреса CSocket активируется до того же самого значения, что и строка символов, представленная в IPXADDR_CHAR, которая введена в качестве второго параметра.

```
Address->ipxa_socket[0]        = 0;
Address->ipxa_socket[1]        = (int) port;

memset(LastAddress,0,sizeof(IPX_ADDR));
 memset(&spx_options,0,sizeof(SPX_OPTS));

struct t_call t_input;
struct t_call t_output;
t_input.addr.maxlen = sizeof(IPX_ADDR);
t_input.addr.len    = sizeof(IPX_ADDR);
t_input.addr.buf    = (char *) Address;
t_input.opt.maxlen  = sizeof(SPX_OPTS);
t_input.opt.len     = sizeof(SPX_OPTS);
t_input.opt.buf     = (char *) & spx_options;
t_input.udata.maxlen = 0;
t_input.udata.len    = 0;
t_input.udata.buf    = NULL;

t_output.addr.maxlen = sizeof(IPX_ADDR);
t_output.addr.len    = sizeof(IPX_ADDR);
t_output.addr.buf    = (char *) LastAddress;
t_output.opt.maxlen  = sizeof(SPX_OPTS);
t_output.opt.len     = sizeof(SPX_OPTS);

t_output.opt.buf     = (char *) & spx_options;
t_output.udata.maxlen = 0;
t_output.udata.len    = 0;
t_output.udata.buf    = NULL;

int rv = t_connect(s,&t_input,&t_output);
```

Данный сегмент кода программы запускает структуру *t_input*, которая является структурой *t_call*, для указания на структуру адреса, созданную на предыдущем шаге. Структура *t_output* активируется таким образом, чтобы адрес сервера был помещен в элемент *LastAddress* из CSocket. Этот CSocket является клиентом, а адрес, к которому вы пытаетесь подключиться, является сервером. Адрес, к которому вы фактически подключились, передается обратно в структуру *t_output*.

```
if ((-1 == rv) && (t_errno == TLOOK) && (T_DISCONNECT == t_look(s)))
 {
```

```
    if (isLogging())
      {
       struct t_discon reason;
       int rv;
       rv = t_rcvdis(s,&reason);
       if (-1 == rv)
         cout << "Error trying to get disconnect reason" << endl;
       else switch(reason.reason)
        { case(TLI_SPX_CONNECTION_FAILED):
            cout << " Connection failed" << endl ;
            break;
          case(TLI_SPX_CONNECTION_TERMINATED):
            cout << " Connection Terminated by Other Side" << endl;
            break;
          case(TLI_SPX_MALFORMED_PACKET):
            cout << " Internal SPX Error — Malformed Packet" << endl;
            break;
         default.
            cout << " Unknown Error: " << reason.reason << endl;
            break;
        }
       return(rv);
      }
  }
```

Данный сегмент кода программы проверяет, отказал ли сервер клиенту путем его отключения в ходе осуществления этапа приема. Если это так, функция соединения не будет выполняться, ошибка будет TLOOK, а в качестве причины функция *t_look* возвратит T_DISCONNECT. В этом случае функция обратится к *t_rcvdis* для получения причины отключения.

```
  if (-1 == rv) return(error("Connecting socket"));
  else flags = flags | 0x04;

  if (isLogging())
   {
cout << "Connecting(" << s  << ") to " << addr << " port " << port << endl;
cout << "Socket is Connected to:" << endl;
sprintf(buf,"  Net:%X %X %X %X\n  Node:%X %X %X %X %X %X\n  Socket: %X %X",
     LastAddress->ipxa_net[0],   LastAddress->ipxa_net[1],
     LastAddress->ipxa_net[2],   LastAddress->ipxa_net[3],
     LastAddress->ipxa_node[0],  LastAddress->ipxa_node[1],
     LastAddress->ipxa_node[2],  LastAddress->ipxa_node[3],
     LastAddress->ipxa_node[4],  LastAddress->ipxa_node[5],
     LastAddress->ipxa_socket[0],LastAddress->ipxa_socket[1]);
cout << buf << endl;
   }
  return(0);
 }
```

Данный сегмент кода программы устанавливает переменные флагов для указания того, что гнездо подключено, и, если разрешен режим регистрации, производится распечатка адреса, к которому это гнездо подключено

```
int CSocket::disconnect()
 {
  if (SOCKET_ERROR == s)
   { if (isLogging())
      cout << "Can't disconnect an error socket" << endl;
     return(-1);
   }
  if (!isConnected())
   { if (isLogging())
       cout << "Not Connected" << endl;
     return(-1);
   }
  int rv = t_snddis(s,NULL);
  if (-1 == rv) return(error("Shutdown... "));
  else flags = flags & 0xFFFB;
  if (isLogging())
    cout << "Connection Terminated" << endl;
  return(0);

 }
```

Данный сегмент кода программы позволяет отключиться от удаленного оконечного пункта, если вы в настоящий момент подключены к нему.

```
int CSocket::recvfrom(char *buffer, int len, int flags)
 {
   if (isLogging())
     cout << "recvfrom not supported on DOS" << endl;
   return(-1);
 }

int CSocket::sendto(char * buffer, int len, int flags,
      IPXADDR_CHAR addr, unsigned short port)
 {
   if (isLogging())
     cout << "sendto not supported on DOS" << endl;
   return(-1);
 }
```

Эти две функции включены для обеспечения совместимости с библиотекой CSockets TCP/IP. В действительности они не реализуются в библиотеке Novell.

```
int CSocket::recv(char *buffer,int len, int flags)
 {
  if (SOCKET_ERROR == s)
   {if (isLogging())
    cout << "Can't recv data over an error socket" << endl;
   return(-1);
   }
  if (!isConnected())
   {if (isLogging())
   cout << "Can't recv data unless you are connected\nSocket:" << s << endl;
   return(-1);
   }
  if (isLogging())
    cout << "Calling recv on socket (" << s << ")" << endl;
```

Большая часть этой функции не меняется в зависимости от реализации TCP/IP. Главным изменением является проверка ошибок, которая требует своего выполнения при обнаружении отключения. Этот сегмент кода не изменяется по сравнению с оригиналом.

```
  int rv = t_rcv(s,buffer,len,&flags);
  if ((-1 == rv) && (t_errno == TLOOK) && (T_DISCONNECT == t_look(s)))
   {if (isLogging())
     { struct t_discon reason;
      int rv = t_rcvdis(s,&reason);
      if (-1 == rv)
       cout << "Error trying to get disconnect reason" << endl;
      else switch(reason.reason)
       { case(TLI_SPX_CONNECTION_FAILED):
          cout << " Connection failed" << endl ;
          break;
         case(TLI_SPX_CONNECTION_TERMINATED):
          cout << " Connection Terminated by Other Side" << endl;
          break;
         case(TLI_SPX_MALFORMED_PACKET):
          cout << " Internal SPX Error — Malformed Packet" << endl;
          break;
        default:
          cout << " Unknown Error: " << reason.reason << endl;
          break;
       }
      return(rv);
     }
   }
  if (-1 == rv) error("receiving data ....");
  if (isLogging())
   {if (T_EXPEDITED & flags) cout << "Expidited data available" << endl;
    cout << "DATA:(" << s << ") len:" << rv << endl;
    output(buffer,rv);
```

```
    if (T_MORE & flags) cout << "More data available..." << endl;
   }
  return(rv);
 }
```

Данный сегмент представляет собой код программы, с помощью которого в действительности осуществляется прием данных и проверка на предмет появления ошибок. При условии разрешенного режима регистрации производится распечатка состояния битов нескольких флагов.

```
int CSocket::send(char *buffer,int len, int flags)
 {if (SOCKET_ERROR == s)
  {if (isLogging())
   cout << "Can't send data over an error socket" << endl;
   return(-1);
  }
  if (!isConnected())
   {if (isLogging())
    cout << "Can't send data unless you are connected\nSocket:"
        << s << endl;
   return(-1);
   }
  if (isLogging())
   cout << "Calling send on socket (" << s << ")" << endl;
```

Подобно компонентной функции *recv* данная функция аналогична реализации TCP/IP.

```
  int rv = t_snd(s,buffer,len,flags);
  if ((-1 == rv) && (t_errno == TLOOK) && (T_DISCONNECT == t_look(s)))
   {if (isLogging())
     {struct t_discon reason;
      int rv;
      rv = t_rcvdis(s,&reason);
      if (-1 == rv) cout << "Error trying to get disconnect reason" << endl;
      else switch(reason.reason)
       { case(TLI_SPX_CONNECTION_FAILED):
           cout << " Connection failed" << endl ;
           break;
         case(TLI_SPX_CONNECTION_TERMINATED):
           cout << " Connection Terminated by Other Side" << endl;
           break;
         default:
           cout << " Unknown Error: " << reason.reason << endl;
           break;
       }
      return(rv);
     }
```

```
    }
  if (-1 == rv) error("send data ....");
  if (isLogging())
    {cout << "DATA:(" << s << ") len:" << len << endl;
     output(buffer,len);
    }
  return(rv);
}
```

Функция **t_send** используется для передачи данных по сети. Процедура проверки ошибок аналогична процедуре проверки при задействовании компонентной функции recv, рассмотренной выше.

```
int CSocket::error(char *data, ...)
 {
      t_error(data);
 return(-1);
 }
```

Функция **t_error** осуществляет распечатку строки символов, за которой следует текст сообщения, характеризующий тип ошибки, которая только что имела место.

ПРИМЕР СЕРВЕРА (С ПОЯСНЕНИЯМИ)

Расхождения между данным сервером и применением TCP/IP выделены жирным шрифтом. Следует отметить, что эти две программы почти идентичны.

```
#include "csocket.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void output(char *, int);

int service(CSocket &);

main(int argc, char *argv[])
 {CSocket::flagdata=0x10;
  CSocket c(-1);
```

Реализация программы (клиент) фирмы Novell зависает во время выполнения *accept*, если два гнезда сети были созданы для обеспечения режима прослушивания. Таким образом создается фиктивное гнездо, пригодное для сохранения значения возврата для функции *accept*, которая будет фактически подключенным гнездом. Значение -1 также превращает его в гнездо ERROR (гнездо ошибки).

```
CSocket s;
```

Это гнездо сервера, находящееся в режиме прослушивания клиентов.

```
int rv;
int port=100;
if (2==argc) port = atoi(argv[1]);

if (!s.isValid())
  { printf("Error Creating initial Socket, program exiting...\n");
    return(-1);
  }
rv = s.bind(port,"00000000:000000000000");
```

Данный адрес запрашивается при привязке, но действительный адрес узла сервера будет привязан к гнезду. Адрес 000000:000000000 является простым фиктивным адресом.

```
if (0!= rv)
  { printf("Error binding the socket to address %d\n",port);
    return(-1);
  }
int quit=0;
while (quit<1)
{c = s.accept();
 while(0==quit)
 {quit = service(c);
 }
}
return(0);
}

int service(CSocket & p)
{
 char buffer[100];
 int i=-1,j=-1;
 char c='A';
 char str[100];
 int rv = p.recv(buffer,100);
 cout << "DATA Recieved on socket (" << p.getSocketValue() << ") Length:" << rv << endl;
 output(buffer,rv);
 if (0==strcmp("quit",buffer)) return(1);
 if (0==strcmp("next",buffer)) return(-1);
 rv = p.send(buffer,rv);
 cout << "Try again, but in format: int string int char " << endl;
```

```
 p >> i >> str >> j >> c;
 cout << "vars:" << i << str << j << c << endl;
 cout << "done" << endl;
 p << "More Data" << i << j << c << str << endl;
 return(0);
}
```

Данная часть сервера аналогична реализации TCP/IP.

ПРИМЕР КЛИЕНТА ФИРМЫ NOVELL (С ПОЯСНЕНИЯМИ)

```
#include "csocket.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void output(char *, int);

main(int argc, char *argv[])
 {
  char *address;
  int rv;
  int port=100;
  if ( (1>=argc) || (3<argc))
   {
    printf("usage: client nnnnnnnn:nnnnnnnnnnnn [socket]\n");
    printf("  nnnnnnnn is the IPX/SPX net of the server\n");
    printf("  :nnnnnnnnnnnn is the IPX/SPX node of the server\n");
    printf("  The server prints this out\n");
    printf("  socket can be used to override the default of 100\n");
    return(0);
   }
```

Вышеприведенный код программы гарантирует передачу клиенту надлежащих аргументов, выдачу сообщения на печать и выход из работы в случае неправильного количества аргументов.

```
 if (2<=argc) address = argv[1];
 if (3==argc) port = atoi(argv[2]);
```

Скопируйте аргументы в соответствующие им переменные.

```
 CSocket::openMode=O_RDWR;
 CSocket::flagdata=0x10;
```

Значение *openMode* по умолчанию для CSocket включает в себя O_NDELAY, что неприемлемо для гнезда **client**. Выйдите из режима

умолчания заданием некоторого значения, не включающего в себя значение O_NDELAY.

Элемент данных флага (flagdata) устанавливается в позицию, разрешающую режим регистрации.

```
CSocket s;
if (!s.isValid())
  { printf("Error Creating initial Socket, program exiting...\n");
    return(-1);
  }

rv = s.bind(port,NULL);
if (0!= rv)
  { printf("Error binding the socket to port %d\n",port);
    return(-1);
  }

s.connect(port,address);
```

Гнездо клиента создается, привязывается к локальному адресу, который еще не определен, а затем подключается к гнезду сервера, которое находится на порте, указанном на узле в определенном адресе.

```
if (0!= rv)
  { printf("Error connecting the socket to address %s\n and port %d\n",
   address,port);
    return(-1);
  }
```

Проверьте успешное завершение подключения.

```
int quit=0;
printf("Type data to send to Server, quit or exit to leave.\n");
printf("   quit also stops the server.\n");
printf("\n");
while (quit<1)
{
 char buffer[1000];

 while(0==quit)
 {
  printf("Client>");
  gets(buffer);
  rv = s.send(buffer,strlen(buffer)+1);
```

```
    printf("%d Characters Sent\n",rv);
    if (-1 == rv) return(0);

    if (0==strcmp(buffer,"quit")) return(0);
    if (0==strcmp(buffer,"exit")) return(0);

    rv = s.recv(buffer,1000);
    printf("%d Characters Recv'd\n",rv);
    if (-1 == rv) return(0);
    output(buffer,rv);

  }
 }
 return(0);
}
```

Код программы, приведенный выше, дает подсказку пользователю ввести данные, передать данные на сервер, а затем получить данные от сервера. Клиент выходит (из цикла) после появления сообщения "quit" (завершение сеанса) или "exit" (выход).

ОПЦИИ КОМПОНОВКИ И КОМПИЛЯЦИИ ФИРМЫ NOVELL

Введение

Несмотря на наличие исходной программы для клиента фирмы Novell, вы можете испытывать некоторые трудности, устанавливая соответствующие опции для компиляции выборки исходного файла. Следующий сегмент детализирует некоторый возможный круг проблем и дает описание их решения.

Версия

Изначально были использованы дискеты размером 3,5 дюйма с версией 1.00(d) для SDK Клиента с ОС NetWare, но заголовочные файлы имели некоторые нарушения целостности, которые были исправлены в дистрибутиве на диске CD-ROM.

Файл Readme.TXT в каталоге NWCLIENT на CD-ROM отображал версию 1.0e, которая является окончательной версией, использованной в примерах. При этом было обнаружено только одно расхождение в заголовочных файлах.

Заголовочные файлы

Файл *CSocket.H* включает в себя следующие заголовочные файлы:

“portable.h”	Пишется пользователем и поставляется в примерах
<windows.h>	Поставляется вместе с Visual C++ или вашим компилятором C++
<iostream.h>	Поставляется вместе с Visual C++ или вашим компилятором C++
“tiuser.h”	Модифицированная копия <tiuser>, поставляемая в SDK фирмы Novell
<“tispxipx.h”>	Поставляется в SDK фирмы Novell
<“nwipxspx.h”>	Поставляется в SDK фирмы Novell

Файл PORTABLE.H определяет следующие константы для примеров ОС NetWare:

```
#ifdef NWDOS
#define u_long unsigned long;
#define SOCKET int
#define SOCKET_ERROR ((SOCKET)-1)
#define INVALID_SOCKET ((SOCKET)-1)
#endif
```

Модификации в tiuser.h

Все проблемы, возникающие в SDK фирмы Novell, находятся в файле “*tiuser.h*”. Основной задачей было создание копии файла в локальном каталоге и осуществление надлежащей модификации файла.

Дискеты 3,5 дюйма с версией 1.00(d) ОС NetWare для клиента SDK

Основной проблемой здесь является то, что прототипы функций, определенные для различных функций TLI в заголовочном файле *tiuser.h*, не являются надежными для языка C++. Следующий алгоритм должен быть использован при определении всех функций:

```
#ifdef _cplusplus
extern “C” {
#endif
define function prototype
define function prototype
define function prototype

#ifdef _cplusplus
}
#endif
```

Такой же алгоритм должен действовать при определении для *t_errno.*

Если вы не осуществите эту модификацию, то вы получите ошибки в LINKER.

CD-ROM с версией 1.00(d) ОС NetWare для клиента SDK

Единственной проблемой в данном случае является то, что определение для идентификатора O_RDWR лежит в пределах *#ifdef* для NWWIN и оно не должно находиться внутри этого условного модуля. Исходным заголовочным файлом, начинающим строку 91, был:

```
#define MAX_TLI_CONNECTIONS 40

/*
 * O_RDWR is defined as different values for various compilers.
 * The compiler used in the TLI DLLs use the following value.
 * This value must be used in any program interfacing with TLI.
 */

#define O_RDWR 0x02

#endif /* NWWIN */
```

Новый файл имеет вид:

```
#endif /* NWWIN */

#define MAX_TLI_CONNECTIONS 40

/*
 * O_RDWR is defined as different values for various compilers.
 * The compiler used in the TLI DLLs use the following value.
 * This value must be used in any program interfacing with TLI.
 */

#define O_RDWR 0x02
```

Данное изменение гарантирует, что два идентификатора — MAX_TLI_CONNECTIONS и O_RDWR — будут всегда определены независимо от того, используется ли программа Windows или нет.

Переменные операционной среды

Следующие две переменные операционной среды должны быть установлены так, чтобы библиотеки SDK могли быть найдены. Переменные операционной среды были установлены в языке Visual C++ с помощью пункта меню Options Directories (каталоги опций).

- Включает в себя путь доступа к файлам (Path): должен быть расширен, чтобы можно было также посмотреть каталог включения SDK.

В нашем случае CD-ROM был подключен к дисководу F:, поэтому было добавлено следующее:

D:\msvc\include;d:\msvc\mfc\include;f:\nwclient\include

- Путь доступа к файлам библиотеки: должен быть расширен для указания на файлы .LIB, которые должны быть с ним связаны:

D:\msvc\lib;d:\msvc\mfc\lib;f:\nwclient\Dos\msc

означает, что вы используете Microsoft C и создаете выполнимую в DOS программу.

D:\msvc\lib;d:\msvc\mfc\lib;f:\nwclient\Dos\bc

означает, что вы используете Borland C и создаете выполнимую в DOS программу.

D:\msvc\lib;d:\msvc\mfc\lib;f:\nwclient\win

означает, что вы создаете выполнимую в Windows программу.

Опции проекта (Project)

Следующие опции проекта были установлены с использованием пункта меню Options Project в Visual C++:

- Тип проекта: выполнимый в DOS
- Опции компилятора: препроцессор: определение символа NWDOS

  модуль памяти: Среда
- Опции компоновщика:: ввод: *mtli*, *mnwipxsp*, *nwnet*
- Файлы в проекте сервера (Server Project):

  CSOCKET.CPP
  TLISERVE.CPP
- Файлы в проекте клиента (Client Project):

  CSOCKET.CPP
  TLICLIEN.CPP

ПРИМЕР СЕАНСА КЛИЕНТА

Запуск передачи данных IPX/SPX у клиента.

Novell IPX/SPX v3.02 Rev. A (901218)
(C) Copyright 1985, 1990 Novell Inc. All Rights Reserved.

```
LAN Option: 3Com 3c503 EtherLink II  v3.01EC (901101)
Hardware Configuration: IO: 300; MEM: C800; IRQ: 3; BNC (Jumpers Config)
```

Прогон модуля клиента

```
C:>TLICLIENT 000000:0080C7D20602

Created socket (0) total sockets:1
 TOpenInfo data
   addr:    12
   options:  8
   tsdu:    -1
   etsdu:   -2
   connect: -2
   discon:  -2
   servtype: 1

Binding(0) to  port 100
Socket is bound to:
   Net:0 0 0 0
  Node:2 60 8C 43 73 5E
  Socket: 40 6

Connecting(0) to 00000000:0080c7d20602 port 100
Socket is Connected to:
   Net:0 0 0 0
  Node:0 80 C7 D2 6 2
  Socket: 0 64

Type data to send to Server, quit or exit to leave.
   quit also stops the server.

Client>Message One

Calling send on socket (0)
DATA:(0) len:12
Message One\000
12 Characters Sent

Calling recv on socket (0)
DATA:(0) len:12
Message One\000
12 Characters Recv'd
Message One\000

Client>23 hello 42 c
Calling send on socket (0)
DATA:(0) len:14
23 hello 42 c\000
14 Characters Sent
```

```
Calling recv on socket (0)
DATA:(0) len:20
More Data2342chello\010
20 Characters Recv'd
More Data2342chello\010
Client>Message Two

Calling send on socket (0)
DATA:(0) len:12
Message Two\000
12 Characters Sent
Message Two\000

Calling recv on socket (0)
DATA:(0) len:12
Message Two\000
12 Characters Recv'd
Message Two\000

Client>47 Goodbye 100 dR
Calling send on socket (0)
DATA:(0) len:17
47 Goodbye 100 d\000
17 Characters Sent

Calling recv on socket (0)
23 Characters Recv'd
More Data47100dGoodbye\010
DATA:(0) len:23
More Data47100dGoodbye\010

Client>quit

Calling send on socket (0)
DATA:(0) len:5
quit\000
5 Characters Sent

Flushing socket (0)
CSocketBuf:Sending
Calling send on socket (0)
DATA:(0) len:0
Closing socket(0)
Deleted socket(0) total sockets:0
```

ПРИМЕР СЕАНСА СЕРВЕРА

```
Starting the IPX/SPX Transport on the server.

Novell IPX/SPX v3.02 Rev. A (901218) (C) Copyright 1985, 1990  Novell Inc. All
Rights Reserved. LAN Option: Xircom Pocket Ethernet II Adapter v1.00EC (910821)
Hardware Configuration: Self Configuring - LPT1:, Bidirectional, IRQ7
Starting the Server

 C:>TLISERVER
 Error Socket Initialized
 Created socket (0) total sockets:1
  TOpenInfo data
    addr:    12
    options: 8
    tsdu:    -1
    etsdu:   -2
    connect: -2
    discon:  -2
    servtype:1
 Binding(0) to 00000000:000000000000 port 100
 Server is bound to:
    Net:0 0 0 0
   Node:0 80 C7 D2 6 2
   Socket: 0 64
 Calling accept to get a client, this may block
 A client is trying to connect
  sequence:  5344
  addr.len:  12
  address:
    Net:0 0 0 0
   Node:2 60 8C 43 73 5E
   Socket: 40 6
   opt.len:  8
 \010\001\000\000\000\000\001\025
   udata.len:0

 Attempting to accept the client
 A new Client is accepted
 Initialized socket (1) total sockets:2
 Error Socket closed
 Socket (-1) being replaced by (1)

 Calling recv on socket (1)
 DATA:(1) len:12
 Message One\000
 DATA Recieved on socket (1) Length:12
 Message One\000
 Calling send on socket (1)
 DATA:(1) len:12
 Message One\000
 Try again, but in format: int string int char
```

```
CSocketBuf:Recving
Calling recv on socket (1)
DATA:(1) len:14
23 hello 42 c\000
vars:23hello42c
done
More Data2342chello\010
CSocketBuf:Sending
Calling send on socket (1)
DATA:(1) len:20
More Data2342chello\010

Calling recv on socket (1)
DATA:(1) len:12
Message Two\000
DATA Recieved on socket (1) Length:12
Message Two\000
Calling send on socket (1)
DATA:(1) len:12
Message Two\000
Try again, but in format: int string int char

CSocketBuf:Recving
Calling recv on socket (1)
DATA:(1) len:17
47 Goodbye 100 d\000
vars:47Goodbye100d
done
More Data47100dGoodbye\010
CSocketBuf:Sending
Calling send on socket (1)
DATA:(1) len:23
More Data47100dGoodbye\010

Calling recv on socket (1)
DATA:(1) len:5
quit\000
DATA Recieved on socket (1) Length:5
quit\000

Flushing socket (0)
sendto not supported on DOS
Closing socket(0)
Deleted socket(0) total sockets:1
Flushing socket (1)
CSocketBuf:Sending
Calling send on socket (1)
DATA:(1) len:0
Closing socket(1)
Deleted socket(1) total sockets:0
```

АЛФАВИТНЫЙ УКАЗАТЕЛЬ

А

Адрес NetWare 29
Адрес сети сервера 394
Адреса IP 42
Адреса XNS 28
Анонимные программные каналы (anonymous pipes) 93
Архитектурная модель фирмы DEC (DNA) 50
Ассоциация (association) 61

Б

Безопасность на уровне разделения в LAN Manager 32
Библиотека TLI 379
Библиотека классов CSocket 217
Блокирующий режим 66, 75
Блоки управления событиями (ECB) 368
Большой эндиан (big endian) 65

В

Ввод/вывод потоков данных 371
Взаимодействие процессов 10
Взаимосвязь открытых систем (OSI) 17
Вилочный (fork()) вызов 91
Виртуальные цепи (virtual-circuit communications) 31, 37
Выводы относительно NetBIOS 39
Вызов mknod() 92
Вызов pipe() 92
Вызов recv() 75
Вызов удаленной процедуры (RPC) 48, 84

Г

Гнезда SPX 368
Гнезда Windows 77
Групповая (multicast) адресация 44
Групповой режим IP 44

Д

Дальний указатель 73
Данные, выходящие за пределы диапазона 75

Дейтаграмма 60, 69
Дейтаграммный протокол пользователя (UDP) 46
Демультиплексировать (demultiplex) 46
Деструктор 218
Динамически компонуемая библиотека (.DLL) 77
Домен BIND 56
Домены операционной системы LAN Manager 32, 56

З

Заголовок (head) потока 371
Загружаемые модули NetWare (NLM) 19
Запрос на комментарий (RFC) 39

И

Именованные программные каналы 91
Именованные программные каналы клиента режима "эхо " (с пояснениями) 251
Инициализация, обработка командной строки 257
Интерфейс NetBIOS с сетью в его отношении к модели OSI 36
Интерфейс транспортного уровня (TLI) 58, 370
Интерфейсные спецификации сетевого драйвера фирмы Microsoft (NDIS) 25
Использование VISUAL C++ 187
Использование языка Visual C++ для построения файлов 265

К

Как работает UDP 48
Клавиатурное эхо в системе "клиент-сервер" 95
Клиент 9
Клиент в режиме "эхо" (с пояснениями) 117
Клиентские библиотеки TCP/IP 101
Клирингхаус (Clearinghouse) 26
Командный терминал (CTERM) 52
Коммуникационный домен (communications domain) 56
Компилирование под NT SDK 139, 187
Компонентная функция unbind 389
Конструктор копирования 218
Конструктор по умолчанию 218
Контрольная сумма 42

Л

Ловушка блокирования 293

М

Малый эндиан (little endian) 65
Межсетевой адрес 44
Межсетевой пакетный обмен (IPX) 18, 25
Метод RPC 86
Методология RPC 85
Модель взаимосвязи открытых систем (OSI) 23, 24
Мультиплексор (multiplexor) потоков 371
Мультирежим системы "клиент-сервер" 167

Н

Новый VISUADOC.CPP (с пояснениями) 319
Новый VISUAVW.CPP 361
Новый VISUAVW.H 358
Номер гнезда 30
Номер порта 46

О

Обзор программирования гнезд 57
Обновление интерфейса пользователя 339
Оконечные пункты связи 58
Оконечные точки передачи данных 370
Операционная система LAN Manager 30
Операционная система NetWare фирмы Novell 18
Описание класса CVISUALDOC 313
Описание класса CVISUALVIEW 357
Описатель (handle) 88
Опции AppWizard 299
Опции компиляции фирмы Novell 405
Опции компоновки фирмы Novell 405
Особенности, требующие внимательного отношения 365
Открытый интерфейс канальных данных (ODI) 25

П

Параллельный сервер (concurrent server) 53, 96
Первая попытка работы в программе-компоновщике 363
Передача потока (stream) символов 48
Перенесение программы клиента в DOS 153
Переопределение оператора (overloading) 219
Переопределение функции (function overloading) 228
Подтверждение приема с повторной передачей 49

Полнодуплексный канал 37
Порядок вызовов TLI 374
Последовательность событий в системе "клиент-сервер" 73
Последовательный обмен пакетами (SPX) 25, 26
Построение готовых к выполнению программ (executables) 187
Построение готовых к выполнению программ с использованием языка Visual C++ 278
Построение программ в SDK 264, 278
Поточные программные каналы (stream pipes) 91
Прерывание 28
Привязывающие описатели (binding handles) 88
Пример клиента фирмы Novell (с пояснениями) 403
Пример сеанса клиента 408
Пример сеанса сервера 411
Пример сервера (с пояснениями) 401
Примеры протоколов 279
Примеры файлов регистрации 266
Пробный шар (ping) 45
Программа многоклиентских именованных программных каналов 269
Программа просмотра 34
Программы обработки события (ESR) 368
Протокол блока сообщений сервера (SMB) 31
Протокол извещения о видах сервиса (SAP) 20, 26
Протокол маршрутной информации (RIP) 20
Протокол межсетевой дейтаграммы (IDP) 24
Протокол определения адреса (ARP) 46
Протокол ошибок (ERROR) 24
Протокол пакетного обмена (PEP) 23
Протокол последовательной передачи пакетов (SPP) 24
Протокол управления передачей (TCP) 48
Протокол управляющих сообщений объединенной сети (Internet) 45
Протокол ядра NetWare (NCP) 26
Протоколы NetWare, отображенные в OSI 27
Протоколы TCP/IP 39
Протоколы XNS 23
Пункт доступа к сервису (SAP) 48
Пятерка 62, 63

Р

Работа с клиентами в мультирежиме (с пояснениями) 170
Работа с серверами в мультирежиме (с пояснениями) 176
Распределенное применение компьютеров в системе "клиент-сервер" (distributed client/server computing) 17
Реализация класса CVISUALVIEW 359
Реализация программы на языке VISUAL C++ 285

Реализация функций USERLIB 349
Регистрация (logging) 158
Регистрация клиента 199
Регистрация сервера (с пояснениями) 188
Редиректор 31
Резидентная программа (TSR) 18

С

Свой (native) канал данных 53
Свойства и наборы свойств сетевой системной базы данных 22
Связь от задачи к задаче 52
Сервер 17
Сервер NPECHO (с пояснениями) 256
Сервер в режиме "эхо" (с пояснениями) 126
Сервис Locator в RPC 88
Сервис Желтых страниц (YP) 21
Сетевая базовая система ввода/вывода (NetBIOS) 27
Сетевая операционная система PATHWORKS 33
Сетевая системная база данных (bindery) 21
Сетевая файловая система (NFS) 33
Сетевой сервисный протокол (NSP) 52
Сетевые именованные программные каналы (networked named pipes) 91
Сетевые операционные системы (NOS) 17
Сетевые транспортные протоколы 9
Система вычисления в открытых сетях (ONC) 88
Система с множественной адресацией (multihomed) 129
Системный вызов 71
Скользящие окна (sliding windows) 49
Служба каталога (directory service) 18
Создание ловушки блокирования 364
Средство обеспечения передачи данных 370
Структура t_info 378
Суперпользователь 47
Схема каталога (Directory Schema) 22

Т

Тип программного канала FIFO 91
Транспортные средства NetBEUI 54
Туннелирование (tunneling) 42

У

Упаковка протоколов TLI фирмы Novell 379
Управление доступом к носителю данных (MAC) 25
Управление каналом данных (DLC) 53
Управление канальным уровнем (LLC) 25
Установки дескриптора файлов (fd_set) 178
Установление связи в программной среде IPC 59
Устройства отображения портов (port mapper) 88

Ф

Файловый сервер (file server) 17
Файл специального устройства (special device file) 92
Фиктивный модуль (stub) 85
Функции NetBIOS 36
Функции протокола IP 42

Х

Характеристики межсетевого протокола (IP) 41
Хорошо известные адреса (well-known addresses) 29, 64
Хорошо известные порты (well-known ports) 29
Хорошо известные порты UDP 47

Ц

Цикл сервера 286

Ш

Шлюзы 41

Э

Эталонная модель ARPANET 41

Я

Язык определений интерфейса Microsoft (MIDL) 87
Язык определения сетевого интерфейса (NIDL) 88

A

accept() 130
ACE (Access Control Elements) 32
AF_DECnet 58
AF_INET 56
ALPHA 33
App Studio 301
AppWizard 298
ARPANET 40
Assignment Operator (оператор присваивания) 218

B

bind 237
bind() 69, 129
Browse 307

C

CCITT 21
cc (команда компилирования/компоновки) 100
ClassWizard 307, 330
connect 171
connect() 72, 121
ConnectNamedPipe 261, 273
CreateNamedPipe 258, 270
CreateThread 274
CSMA/CD 51
CSOCKET.CPP (с пояснениями) 226
CSOCKET.CPP фирмы NOVELL (с пояснениями) 379
CSOCKET.H с пояснениями 220
CVisualDoc 309
CVisualView 309

D

DARPA Internet 40
DECnet 50
DECnet-DOS 52
DECNODE.DAT 84
DNP.EXE 53
DOS_SOCK.LIB 101

E

Ethernet II 51

F

FIFO 91
FIONREAD 77, 105
fork 53

G

gethostbyname() 84
getnodebyname() 84

H

htons() 105

I

IEEE — Институт инженеров по электротехнике и радиоэлектронике 28
IEEE 802.2 SNAP 27
IEEE 802.3 27
IEEE 802.5 27
inet_addr() 104
INT 21H 28
InterProcess Communications (IPC) 55
inUseFlag 369
ioctl() 76, 101
iostream 219
IOSTREAM.H 217
IPX (межсетевой пакетный обмен) 18, 25
IPX.COM 19, 373
IPXOpenSocket() 369

L

LAN Manager 31, 37
listen() 70

M

ManageWORKS 34
Microsoft Networks (MS-NET) 33

MSG_OOB 75, 112
MSG_PEEK 75

N

native 53
NetBEUI 9
NetBIOS 35, 37
netstat 72
NetWare 18, 28
NetWare V.4x 21
NETX.COM 19
NMAKE.EXE 106

O

open() 56
OpenVMS 33
OSF/1 33
OSLIB 20

P

Packet Internet Grouper 45
PATHWORKS 33
версия 5.0 33
Версия 5.1 PATHWORKS 34
pipe(), 91
port mapper 88
Portmapper 29
prgpre.h 108
PRGPRE.H 79, 82
Protocol Family_Internet (PF_INET) 56
PWIMPORT.LIB 82
PWNODE.DAT 84
PWSOCK.DLL 82

Q

QuickWin 154

R

recv() 66, 125, 131
recvfrom() 73, 76
RPC Locator 88

S

sclose() 112
Select 292
select() 181
send() 73, 76, 124
SENDDIALOG 327
sendto() 73
SETHOST.EXE 52
setsockopt() 112
shutdown 240
sioctl() 66
SOCK_SEQPACKET 69
SOCK_STREAM 68
SOCKADDR 122
SOCKADDR_IN 122
socket() 56, 59, 68
SO_KEEPALIVE 112
sread() 73, 75
STDIO.H, STDLIB.H 118
swrite() 67

T

t_accept 391
t_alloc 377
t_bind 376
TELNET 52, 57
t_free 377
t_getinfo 377
t_listen 390
t_open 377
t_output 389
t_send 401

U

ULTRIX 57
USERLIB.C 186
USERLIB.C (с пояснениями) 133
UserLib.h 170

V

VAX-кластеры 34
VISUADOC.CPP (с пояснениями) 332

VISUADOC.H (с пояснениями) 314
Visual C++ 139

W

WaitNamedPipe 250, 254
Windows NT 33, 115
Windows V3.1/DOS 115
WINSOCK 55, 115
WINSOCK.DLL 63, 78
WINSOCK.H 81, 118, 170
WINSOCK.LIB 78
WSACleanup() 81
WSAStartup 120, 287
WSAStartup() 81
WSOCK32.LIB 139

X

Xerox Courier 88
XNS 28

Эд Барфилд, Брайен Уолтерс

Программирование «клиент-сервер» в локальных вычислительных сетях

Перевод с английского Г.П. Фролов

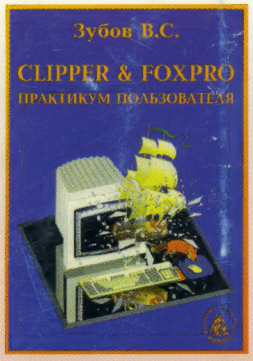
Редактор Е.А. Черноиванова
Корректор О.Н. Богачева
Художник Е.Б. Кафтырева
Подготовка оригинал-макета Н.П. Рогатко

Сдано в набор 02.04.97. Подписано в печать 26.05.97.
Формат 70×100/16. Бумага газетная. Гарнитура Times.
Печ. л. 26,5. Тираж 5000 экз. ЛР № 063643 от 14.10.94. Заказ № 6711

Отпечатано с оригинал-макета в филиале Государственного ордена Октябрьской революции, ордена Трудового Красного Знамени Московского предприятия «Первая Образцовая типография» Комитета РФ по печати. 113114, Москва, Шлюзовая наб.,10

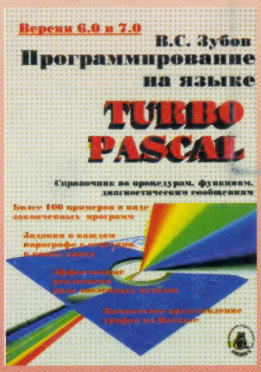
Информационно-издательский дом «ФИЛИНЪ»
г. Москва, ул. Новопетровская, д. 1, стр. 7
тел./факс: (095) 459 41 31, 450 00 41

Информационно-издательский дом "ФИЛИНЪ" предлагает Вашему вниманию следующие книги компьютерной тематики:

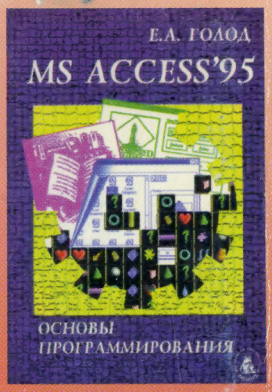


Зубов В.С.
CLIPPER & FOXPRO
Практикум пользователя

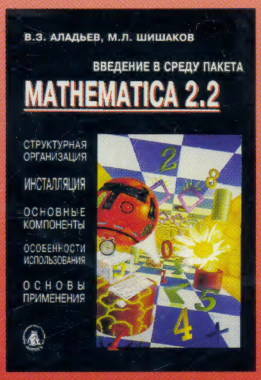
Дубина А.
Сынгур С.
FOX PRO 2.x
Методы программирования



Зубов В.С.
Программирование на языке
TURBO PASCAL



Голод Е.А.
MS ACCESS'95.
Основы программирования



Аладьев В.З.
Шишаков М.Л.
Введение в среду пакета
MATHEMATICA 2.2

ISBN 5-89568-015-1